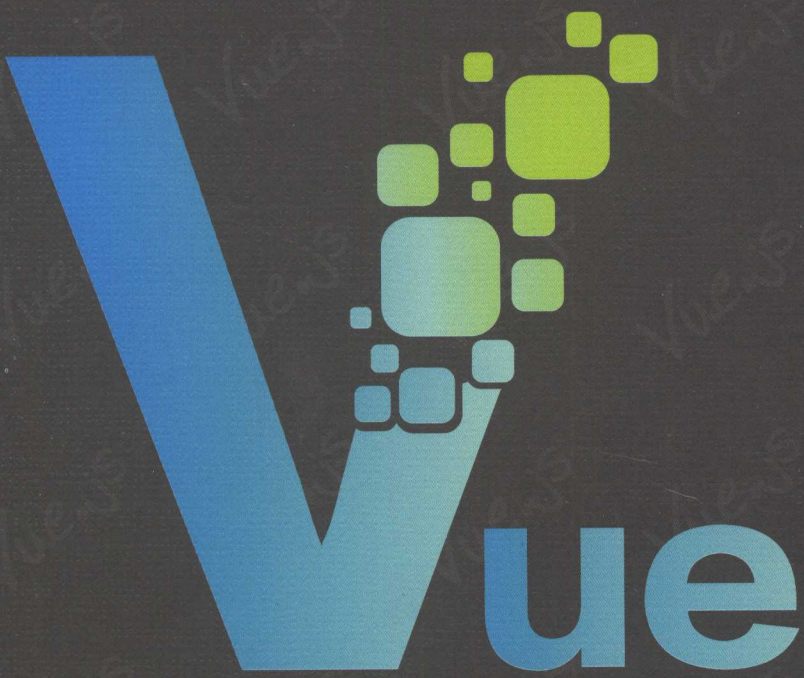
Vue 前端工程师必备技能

Vue 移动开发实战技巧

在线疑难解答
前端案例资源
读者圈

李利德 徐辛承 编著



中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

前端工程师必备技能
Vue 移动开发实战技巧

李利德　徐辛承　编著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

Vue（Vue.js）是一个渐进式的 JavaScript 框架，与其他重量级框架不同的是，Vue 采用自下而上增量开发的设计。Vue 的核心库只关注视图层，它不仅易于上手，还便于与第三方库或既有项目整合。作为 2016 年社区最火的前端框架，越来越多的公司都在尝试用 Vue 来开发自己的项目。

本书主要从项目维度出发，站在实战的角度，从项目搭建、项目开发到项目优化，结合各种实用 demo，结合开发环境构建，从无到有，剖析原理，全面介绍 Vue2.0 的实用技巧。书的第四章重点讲解 Vue 内部实现机制，针对各种业务形态的支持以及网站调优方案，等等，是 Vue 技术体系追随者不可多得的实战宝典。

读者还可以通过在线服务平台获得作者提供的在线学习资源和以及疑难解答。

本书适合软件开发人员、前端工程师、计算机专业大学生学习参考。

图书在版编目（CIP）数据

前端工程师必备技能：Vue 移动开发实战技巧 / 李利德，徐辛承编著. —北京：电子工业出版社，2018.1

ISBN 978-7-121-33156-5

Ⅰ. ①前… Ⅱ. ①李… ②徐… Ⅲ. ①网页制作工具－程序设计 Ⅳ. ①TP392.092.2

中国版本图书馆 CIP 数据核字（2017）第 298353 号

策划编辑：张瑞喜
责任编辑：张瑞喜
印　　刷：中国电影出版社印刷厂
装　　订：中国电影出版社印刷厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编　100036
开　　本：710×1000　1/16　印张：18　字数：323 千字
版　　次：2018 年 1 月第 1 版
印　　次：2018 年 1 月第 1 次印刷
定　　价：58.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：zhangruixi@phei.com.cn。

序言

在很长时间以内，前端开发的工作内容是为静态的 HTML 增加动态的效果，所以基于 jQuery 的 Bootstrap 非常流行。真正让前端开发产生变化的是 2014 和 2015 年 Angular.js 和 React 的崛起，这使得 MV*模式变得流行，让前端渲染开始流行。MV*模式让前端开发的思维发生了一个翻天覆地的变化，让前端工作变得更有挑战，它不再是为后端开发者写 HTML 的模板，而是更多地关注在工程化、可维护性、数据流等方面。

“饿了么”大前端团队在 2015 年主要使用的是 Angular.js 1.2 版本，在使用其开发了外卖 PC 站、早餐业务移动站之后，我们体会到了 Angular.js 的一些痛点。主要的痛点是组件封装的不便、一些历史遗留的设计问题和性能问题，所以我们开始尝试一些新的解决方案。在 2015 年的下半年，Vue.js 发布 1.0 之后，我们尝试使用 Vue.js 来完成一些业务的开发。当时 Vue.js 生态还不是很成熟，我们就边写业务边做了一些组件发布到了开源社区。

当初我们选用 Vue.js 1.0 的主要原因是团队比较熟悉 Angular.js，在我看来 Vue.js 1.0 版本还只是 Angular.js 1.x 的改良版。但是在 2016 年初，Vue.js 2.0 公开之后，我觉得 Vue.js 2.0 将会成为能与 React 竞争的现代前端框架，所以决定基于 Vue.js 为团队做一些基础设施建设。

我们团队先后开发了移动端和 PC 端的两套前端组件库：Mint UI 和 Element UI，并在开发完成之后发布到了开源社区，为 Vue.js 的社区活跃做出了一些贡献，也在开源社区取得了一些影响。截止目前，Element UI 仍然是 Vue.js 社区中 Star 最多、最活跃、最成熟的组件库，我可以负责任地说，Element UI 一直应用在我们的生产项目中。

回顾加入饿了么大前端以来的两年多里，推动团队选用 Vue.js 作为前端框

架可能是我做的最明智的决定。当初放弃 Angular.js 选择 Vue.js 是个艰难的决策，幸运的是我们能在这段时间里与 Vue.js 社区共同成长。

如果你是初学者，想入门 Vue.js，那么本书可以让你充分体会 Vue.js 的魅力；如果你想对 Vue.js 有更为深入的了解，本书对 Virtual DOM 和 Vue Router 的原理解析应该对你有所帮助。最后，本书涵盖了 Vue.js 2.1、2.3 的一些新特性，还对阿里巴巴的开源项目 Weex 进行了一定的讲解，这保证了本书具有较新的时效性。在细读了一遍后，我觉得这本书适合各个阶段的前端开发者，在此真诚地推荐给大家。

张　龙

饿了么高级研发经理&Element UI 创始人

前言

前端框架的发展

写这本书之前，思考过一段时间，最后问题回归到 Vue 是什么？从哪来？到哪去？然后，想到了前端框架的发展，当然在这里不敢妄谈演进，只说一下本人对前端的认识过程。

本人是 2008 年接触前端开发的，那个时候中国公司极少数存在前端开发这个岗位，很多地方还叫网页开发，大部分还是从数据存储到页面展示一把抓。这个时期，以能将文本文档写 html、css、js 为大牛的标准，你知道的细节越多，意味着你越高深，当然这是学生阶段，大家都讲求一个装字。之后接触一些 js 框架，诸如 prototype、mootools、jquery 等是这个时代的主流，后来 jquery 因为它的易用，高效异军突起，占据主导；这个时代框架主要给大家解决的是兼容性的处理和代码的简化。

其次，随着电脑性能、浏览器支持及网速的不断攀升，前端页面越来越像一个桌面应用，各种交互功能需要在页面中实现，就产生了各种 mv*的框架，如 backbone、AngularJS 等，它们要解决的问题也是很有针对性的，让界面、操作、数据分开，在复杂的交互中，找到一条不变的原则。

再次，随着 Nodejs 的兴起以及前端环境的进一步发展，前端模块化的概念得到广泛的发展，这段时间与上一段内容并非完全的分先后，有部分重叠；就本人而言，最后解除模块化，模块化的出现也是适应前端工程化进展的重要标志。从一个或者几个文件就能完成一个页面，到一个页面需要一个工程；这种变化也意味着前端进入了技术壁垒，从此，后端开发者（php，java）已经很难上手一个前端项目。

最后，mvvm 框架模型概念火爆起来，react 首先为大家熟知，并迅速传播火热，随后 Vue 以其灵活性高、开发效率高等进入大家视野，占得一席之地；是 react 好还是 Vue 更好，也成为人们津津乐道的话题，至于答案，我认为适合的就是最好的。

本书面向的人群

本书前五章内容涉及基础开发环境的搭建和API的介绍及开发过程中遇到的问题，书中的代码是对Vue官方文档里样例的扩展，对于初步使用Vue的人群有很好的指导意义。后五章内容是实战的技巧及原理的剖析，适用对框架有一定经验、对原理有研究的人参考。

本书由长期从事Vue开发和研究的百度外卖研发团队撰写完成，充分结合实际，紧跟Vue.js最新特性，对Vue.js作为技术栈的架构有深入的理解，并在此基础上搭建了符合实际业务场景的整套技术栈，对于想重构已有工程的团队，也有很好的指导作用。

Vue适用的场景

大量的项目与实践表明，Vue可以适用于各个场景。我们的Vue用在H5运营后台、离线组件等各种业务形态中，最近所做的ssr也是以Vue为基础来做的。我所在的团队以业务为先，没有独立的基础技术部门，所有技术尝试都是直接作用于线上业务项目中，在进行过多次尝试改造后，慢慢形成了一种以Vue为主的架构体系，配合我们自己封装的构建工具fekey，组合成一套前端全栈解决方案；尤其在运营后台项目中，形成了一套快速搭建页面的框架（BLOCKS），人力工效提升5～8倍。当然Vue并不适用于所有项目，具体项目具体决策，受成本、时间、团队的影响，选择自己合适的框架进行开发，这也是一个团队技术领袖应该具有的能力。

在线学习与疑难解答

读者圈

本书特别为广大读者精心准备了专有的在线服务与交流平台。用微信扫描右侧二维码，关注“悦读力”，即可加入本书读者圈，获得关于前端技术相关的更多学习资源，并有机会向本书作者李利德老师提问，得到在线学习指导。

这是我们这个开发团队第一次写书，参加本书编写人员还有白杨、李雅男、董庆明、陈立文、肖仁晖。书中表达与样例难免会有纰漏和谬误，如果发现，欢迎联系我们反馈，万分感谢！我们的邮箱是：lld490112728@163.com。

李利德

百度外卖高级技术经理

目录

读 者 圈

第 1 章　搭建开发环境

本章主要介绍搭建本地 Node 环境以及 ide 的一些常用配置。

1.1　本地 Node 环境

1.1.1　Node.js

1.1.1.1　Node 介绍

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。

Node.js 使用一个事件驱动、非阻塞式 I/O 的模型，使其轻量又高效。

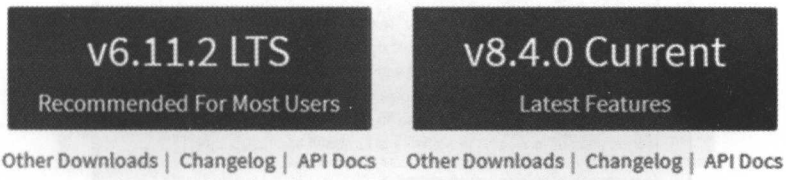
Node.js 的包管理器 npm，是全球最大的开源库生态系统。

1.1.1.2　安装步骤

1.　打开 Node

打开 Node 官网，往下拉，就可以看到有两个下载选项。

如图 1-1 所示，图 1-1（a）是 LTS 版本，就是长期支持的稳定版本，推荐使用这个版本。



（a）　　（b）

图 1-1

图 1-1（b）是最新版本，支持更多的语言特性，对 es6 有更多的支持（可以在 http://Node.green 上面获取到 Node.js 各个版本对 es6 的支持情况）。

根据自己的需要选择版本下载。

2. 打开安装包

打开下载好的安装包，出现安装界面，点击 Next，如图 1-2 所示。

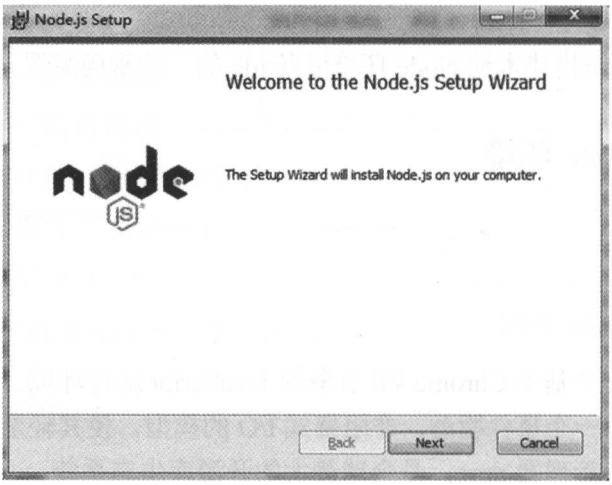


图 1-2

在接下来的窗口中，选中接受许可证协议，然后点击 Next，如图 1-3 所示。

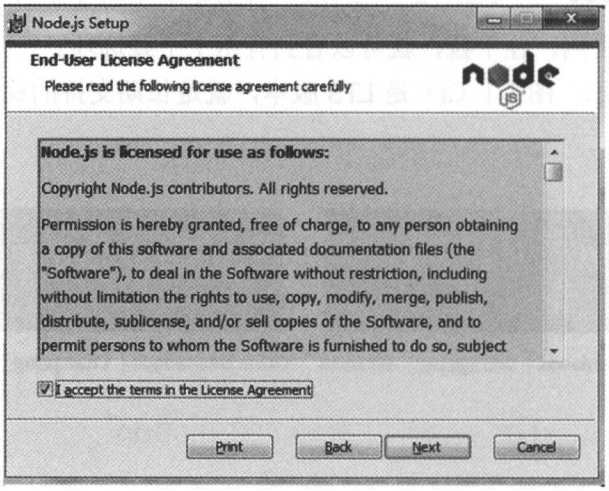


图 1-3

选择安装的目录，如图 1-4 所示。

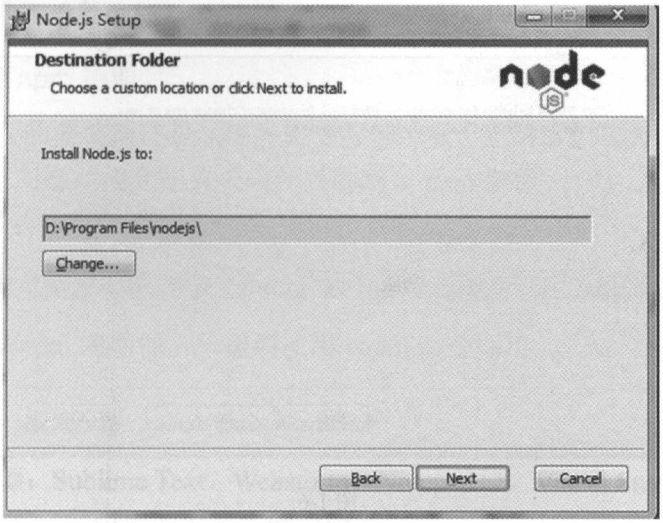


图 1-4

点击 Next，如图 1-5 所示。

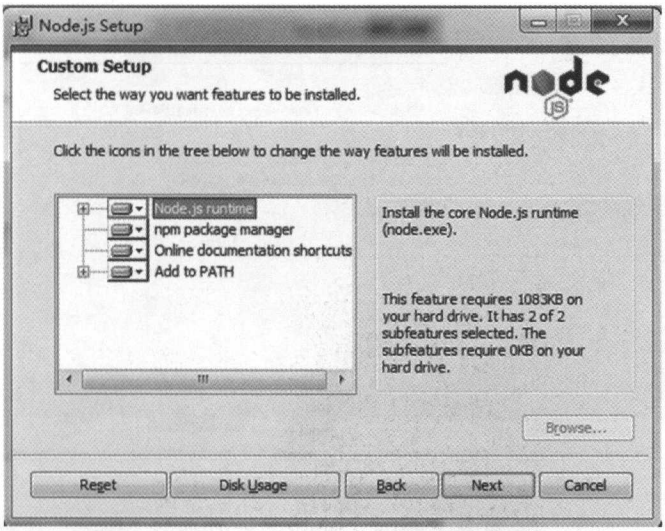


图 1-5

点击 Install，开始安装，如图 1-6 所示。

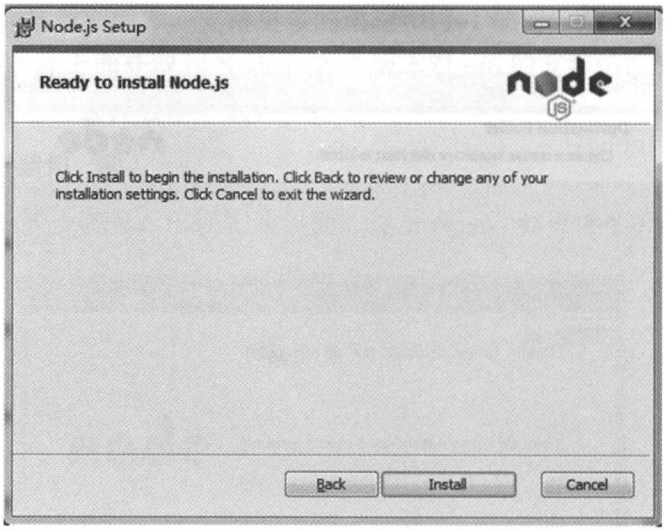


图 1-6

点击 Finish，安装完成，如图 1-7 所示。

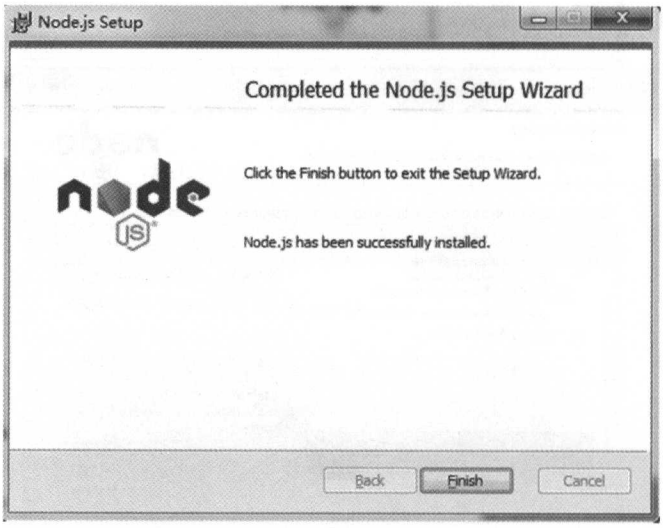


图 1-7

3. 验证

验证：打开终端输入下面的命令，就可以验证自己的 Node 和 npm 是否安装成功。

```
Node -v
npm -v
```

4. 淘宝 npm 镜像

由于 npm 的安装插件是从国外的服务器下载，受网络影响大，经常会出现下载缓慢或异常问题，因此建议采用国内的淘宝 npm 镜像。

安装命令：

```
npm install -g cnpm --registry=https://registry.npm.taobao.org
```

之后安装 npm 的插件，只需要使用 cnpm 命令即可。

1.1.1.3　ide 配置

ide 编辑器：Sublime Text，Webstorm，Notepad++，Visual Studio Code 等。主要以 Sublime Text 编辑器的安装为例说明。

把 build 环境设置为 Nodejs，如图 1-8 所示。

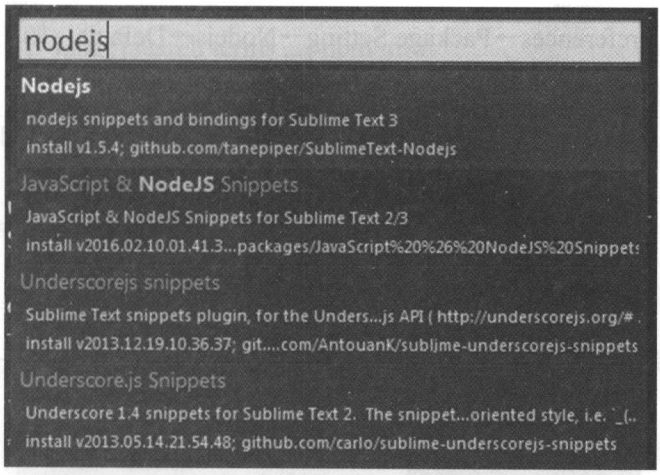


图 1-8

菜单→Tools→Build System→Nodejs，如图 1-9 所示。

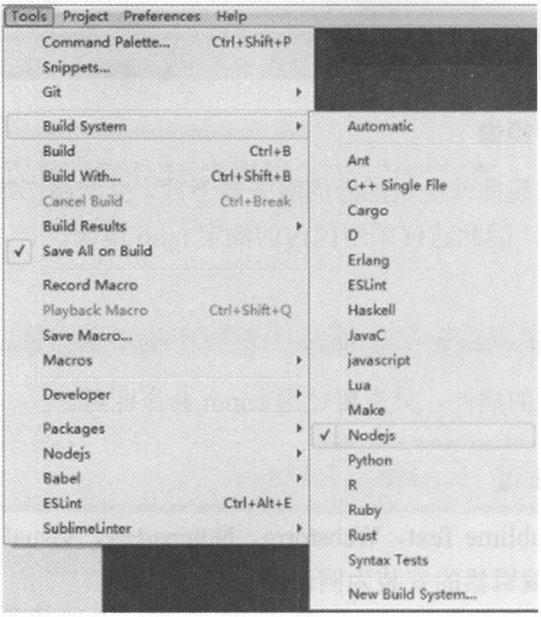


图 1-9

配置环境 Preferences→Package Setting→Nodejs→Default，如图 1-10 所示。

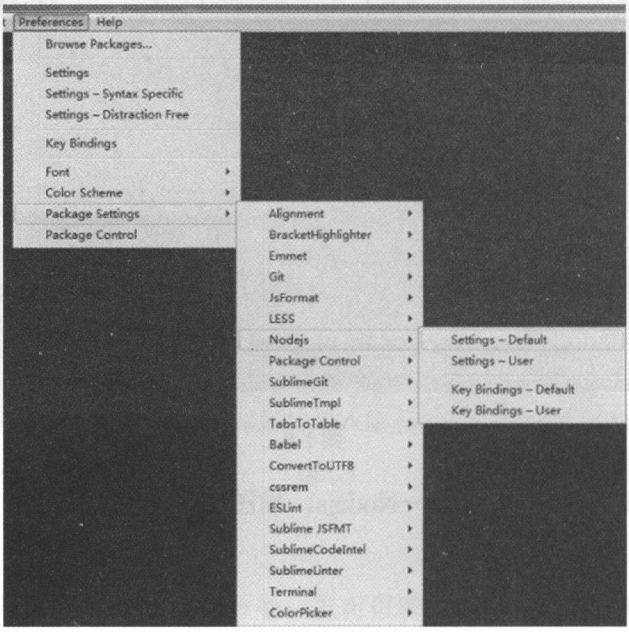


图 1-10

Node.js 的包管理工具有 npm 和 yarn 等。

1.1.2　npm

1.1.2.1　npm 介绍

npm 全称是 Node package manager，即 Node 的包管理器，用于 Nodejs 的插件管理（包括安装、卸载、管理依赖等）。

1.1.2.2　npm 常用命令

获取输入命令的详细信息：

npm help <term>；

查看各命令的简单用法。

npm -l；

初始化 package.json；

npm init；

搜索与查询：

npm search [--long] [search terms ...] npm info。

全局安装：

npm install –g。

删除：

npm uninstall。

更新：

检查当前安装的所有 npm 包是否有更新；

先使用 npm outdated 来检查当前安装的所有 npm 包是否有更新。检查更新如图 1-11 所示。

更新时，可以使用 npm update 命令来更新所有可更新包，如果只想更新某个具体的包（以 Vue 为例），只需要输入如下命令：

npm update Vue

发布，如果你想发布一个包到 npm，首先你需要注册一个账号，可以使用 npm adduser 命令，或者在网站注册。

然后，使用 npm publish 来指定一个目录发布：

npm publish。

```
$ npm outdated
Package                          Current  Wanted  Latest  Location
autoprefixer                       6.7.2   6.7.6   6.7.6  wmui
babel-core                        6.22.1  6.23.1  6.23.1  wmui
babel-loader                      6.2.10   6.3.2   6.3.2  wmui
babel-plugin-component             0.7.0   0.7.0   0.9.0  wmui
babel-plugin-transform-runtime    6.22.0  6.23.0  6.23.0  wmui
babel-register                    6.22.0  6.23.0  6.23.0  wmui
css-loader                        0.25.0  0.25.0  0.26.2  wmui
element-ui                         1.1.6   1.2.3   1.2.3  wmui
express                           4.14.1  4.15.0  4.15.0  wmui
extract-text-webpack-plugin        1.0.1   1.0.1   2.0.0  wmui
file-loader                        0.9.0   0.9.0  0.10.1  wmui
http-proxy-middleware             0.17.3  0.17.4  0.17.4  wmui
ora                                0.3.0   0.3.0   1.1.0  wmui
postcss-pxtorem                    3.3.1   3.3.1   4.0.0  wmui
sortablejs                         1.4.2   1.5.1   1.5.1  wmui
url-loader                         0.5.7   0.5.8   0.5.8  wmui
vue                               2.1.10   2.2.1   2.2.1  wmui
vue-loader                         9.9.5   9.9.5  11.1.4  wmui
vue-style-loader                   1.0.0   1.0.0   2.0.3  wmui
vuedraggable                       2.4.0   2.8.5   2.8.5  wmui
vuex                               2.1.2   2.2.1   2.2.1  wmui
webpack                           1.14.0  1.14.0   2.2.1  wmui
webpack-dev-middleware            1.10.0  1.10.1  1.10.1  wmui
webpack-hot-middleware            2.16.1  2.17.1  2.17.1  wmui
webpack-merge                     0.14.1  0.14.1   3.0.0  wmui
```

图 1-11

1.1.2.3 npm link

对开发者而言，这是最有价值的命令。假设我们开发了一个模块叫 test，然后我们在 test-example 里引用这个模块，每次 test 模块的变动我们都需要反映到 test-example 模块里。使用 npm link 命令一切变得非常容易。

首先，我们需要把 test 链接到全局模式下：

```
cd ~/work/Node/test #进入 test 模块目录
npm link #创建链接到 \$PREFIX/lib/Node_modules
```

其次，test 的模块将被链接到$PREFIX/lib/Node_modules 下，就像我的机器上 $PREFIX 指到/usr/local，那么/usr/local/lib/Node_modules/test 将会链接到~/work/Node/test 下。执行脚本 bin/test.js 被链接到/usr/local/bin/test 上。

接下来，我们需要把 test 引用到 test-example 项目中来：

```
cd ~/work/Node/test-example #进入 test-example 模块目录
npm link test #把全局模式的模块链接到本地
```

npm link test 命令会去$PREFIX/lib/Node_modules 目录下查找名叫 test 的模块，找到这个模块后，把 $PREFIX/lib/Node_modules/test 的目录链接到 ~/work/Node/test-example/Node_modules/test 这个目录上来。

现在任何 test 模块上的改动都会直接映射到 test-example 上来。比如，假设我们开发很多应用，每个应用都会用到 Coffee-script：

```
    npm install coffee-script -g #全局模式下安装 coffee-script
    cd ~/work/Node/test #进入开发目录
    npm link coffee-script #把全局模式的 coffee-script 模块链接到本地的
Node_modules 下
    cd ../test-example #进入另外的一个开发目录
    npm link coffee-script #把全局模式的 coffee-script 模块链接到本地
    npm update coffee-script -g #更新全局模式的 coffee-script，所有 link 过去
的项目同时更新了。
```

npm link 对于开发时一个模块被多个模块引用是非常有用的。只要 Node.js 支持 fs.symlink，就可用到这个特性。

1.1.2.4　npm scripts

npm 不仅可以用于模块管理，还可以用于执行脚本。package.json 文件中有一个 scripts 字段，可以用于指定脚本命令，提供给 npm 直接调用。例如：

```
  "scripts": {
    "dev": "Node build/dev-server.js",
    "build": "Node build/build.js"
  }
```

执行 npm run dev，相当于执行 Node build/dev-server.js 命令。

npm run 是 npm run-script 的缩写，一般都是用前者，但是后者可以更好地反映这个命令的本质。直接执行 npm run 或 npm run-script，不加参数会列出 scripts 属性下所有可运行的命令及其参数。

1.1.2.5　package.json 文件

package.json 对于 npm 来说是非常重要的元数据集，对于我们的应用来说，package.json 文件使用 dependencies 和 devDependencies 来定义应用依赖和开发环境依赖。对于一个 web 应用来说，你的自动构建任务所依赖的包，属于开发环境依赖。如果你的 web 应用依赖 jQuery 来发起 Ajax 请求，它应该属于应用依赖。

package.json 文件可以用手工编写，也可以使用 npm init 命令自动生成。

package.json 文件内部就是一个 json 对象，该对象的每一个成员就是当前项目的一项设置。

package.json 中最重要的属性是 name 和 version 两个属性，这两个属性是必须要有的，否则模块就无法被安装，这两个属性一起形成了一个 npm 模块的唯一

标识符。模块中内容变更的同时，模块版本也应该一起变化。

name 属性就是你的模块名称，下面是一些命名规则：

```
name 必须小于等于 214 个字节，包括前缀名称在内（如 xxx/xxxmodule）。
name 不能以"_"或"."开头
不能含有大写字母
name 会成为 url 的一部分，不能含有 url 非法字符
```

下面是官网文档的一些建议：

```
不要使用和 Node 核心模块一样的名称
name 中不要含有"js"和"Node"。
name 属性会成为模块 url、命令行中的一个参数或者一个文件夹名称，任何非 url 安全的字符在 name 中都不能使用，也不能以"_"或"."开头
name 属性也许会被写在 require()的参数中，所以最好取个简短而语义化的值。
```

version 是版本。version 必须可以被 npm 依赖的一个 Node-semver 模块解析。

scripts 指定了运行脚本命令的 npm 命令行缩写。scripts 属性是一个对象，里边指定了项目的生命周期各个环节需要执行的命令。key 是生命周期中的事件，value 是要执行的命令。

dependencies 字段指定了项目运行所依赖的模块，devdependencies 指定项目开发所需要的模块。它们都指向一个对象。该对象的各个成员，分别由模块名和对应的版本要求组成，表示依赖的模块及其版本范围。版本范围是一个字符，可以被一个或多个空格分割。对应的版本可以加上各种限定，主要有以下几种：

```
指定版本：比如 1.2.2，遵循“大版本.次要版本.小版本”的格式规定，安装时只安装指定版本。

波浪号+指定版本：~1.2.2，表示安装 1.2.x 的最 新版（不低于 1.2.2）。但是不安装 1.3.x，也就是说安装时不改变大版本号和次要版本号。

插入号+指定版本：^1.2.2，表示安装 1.x.x 的最新版（不低于 1.2.2）。但是不安装 2.x.x，也就是说安装时不改变大版本号。需要注意的是，如果大版本号为 0，则插入号的行为与波浪号相同，这是因为此时处于开发阶段，即使是次要版本号变动，也可能带来程序的不兼容。

latest：安装最新版本。
```

dependencies 也可以被指定为一个 git 地址或者一个压缩包地址。

devdependencies 主要用于指定在开发过程中，使用的额外测试或者文档框架。这些模块会在 npm link 或者 npm install 的时候被安装，也可以像其他 npm 配置一

样被管理。

对于一些跨平台的构建任务，例如，把 CoffeeScript 编译成 JavaScript，就可以通过在 package.json 的 script 属性里边，配置 prepublish 脚本来完成这个任务，然后，需要依赖 coffee-script 的模块写在 devdependencies 属性中。

```
{
  "name": "ethopia-waza",
  "description": "a delightfully fruity coffee varietal",
  "version": "1.2.3",
  "devDependencies": {
    "coffee-script": "~1.6.3"
  },
  "scripts": {
    "prepublish": "coffee -o lib/ -c src/waza.coffee"
  },
  "main": "lib/waza.js"
}
```

prepublish 脚本会在发布之前运行，因此，用户在使用之前就不用再自己去完成编译的过程了。在开发模式下，运行 npm install 也会执行这个脚本，因此可以很方便的调试。

如果一个模块不在 package.json 文件之中，可以单独安装这个模块，并使用相应的参数，将其写入 package.json 文件之中。

```
npm install express --save

npm install express --save-dev
```

上面代码表示单独安装 express 模块，--save 参数表示将该模块写入 dependencies 属性，--save-dev 表示将该模块写入 devdependencies 属性。

peerdependencies。

有时候做一些插件开发，比如 grunt 等工具的插件，它们往往是在 grunt 的某个版本的基础上开发的，而在它们的代码中并不会出现 require("grunt")这样的依赖，dependencies 配置里边也不会写上 grunt 的依赖，为了说明此模块只能作为插件运行在宿主的某个版本范围下，可以配置 peerdependencies：

```
{
  "name": "tea-latte",
  "version": "1.3.5"
  "peerDependencies": {
```

```
    "tea": "2.x"
  }
}
```

上面这个配置确保在 npm install 的时候，tea-latte 会和 2.x 版本的 tea 一起安装，而且它们两个的依赖关系是同级的：

```
├── tea-latte@1.3.5
└── tea@2.2.0
```

这个配置的目的是让 npm 知道，如果要使用此插件模块，请确保安装了兼容版本的宿主模块。

1.1.3 yarn

1.1.3.1 yarn 介绍

yarn 是 Facebook，Google，Exponent 和 Tilde 开发的一款新的 JavaScript 包管理工具。它的目的是解决使用 npm 面临的一系列问题，即保证安装时速度、一致性和安全问题（npm 安装时允许运行代码）。Fast, reliable, and secure dependency management.极速、可靠、安全的依赖管理。

优点：yarn 缓存它下载的每个包，所以无需重复下载。它还并行化操作以最大化资源利用，所以安装时间比以往快。

yarn 能够更快地安装软件包和管理依赖关系，并且可以在跨机器或者无网络的安全环境中保持代码的一致性。

yarn 提高了开发效率，并解决了共享代码时面临的一些问题，使得工程师们可以专注在构建新产品以及新特性上。

用 yarn 来安装依赖比较快，先全局安装 yarn，然后用 yarn 命令来代替。

1.1.3.2 yarn 安装

windows 安装：

参考 https://yarnpkg.com/en/docs/install#windows-tab。

安装步骤：

打开下载的.msi 文件，点击 Next，如图 1-12 所示。

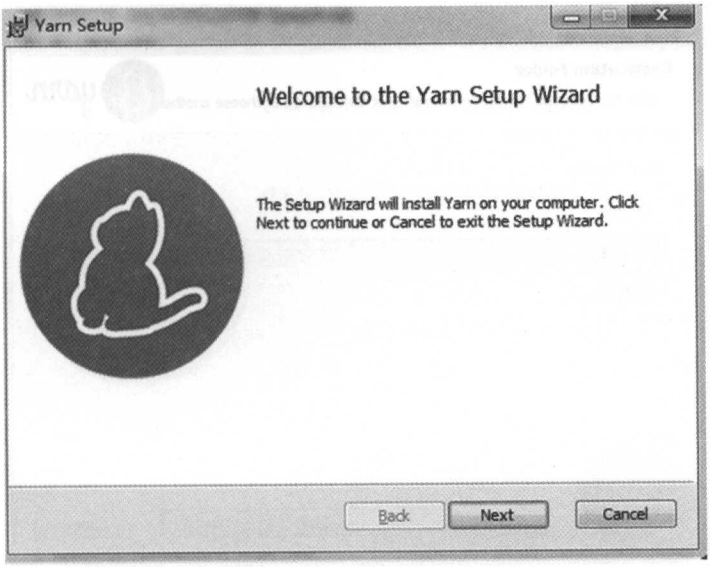


图 1-12

选中接受许可证协议的复选框，点击 Next，如图 1-13 所示。

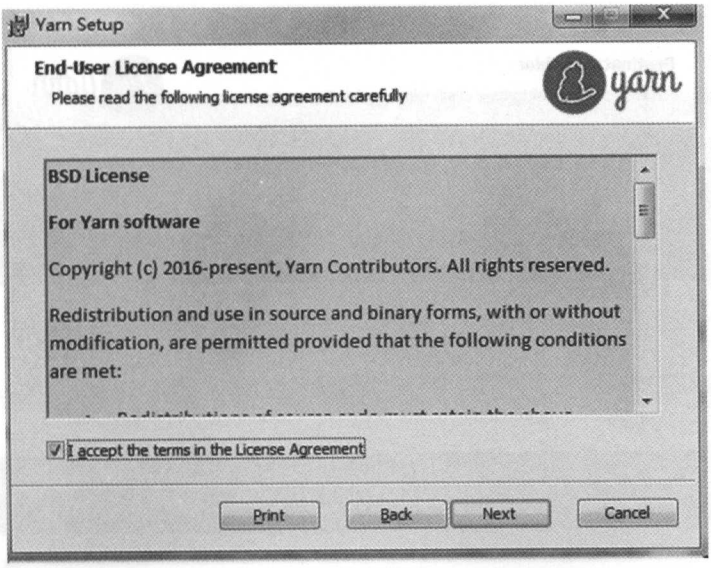


图 1-13

选择安装目录，如图 1-14 所示。

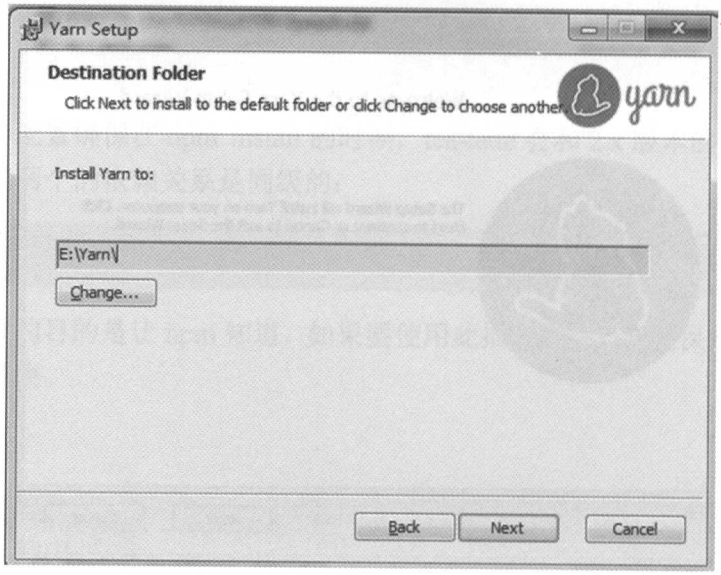


图 1-14

点击 Install，如图 1-15 所示。

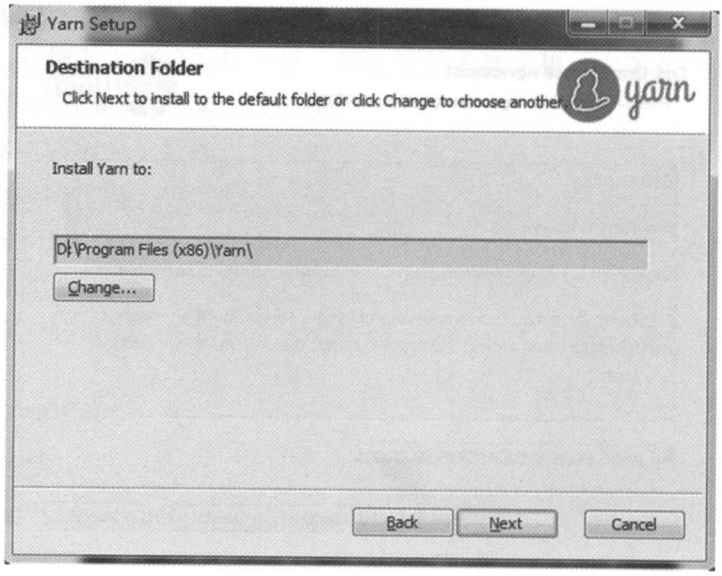


图 1-15

安装完成后，点击 Finish，如图 1-16 所示。安装成功。

图 1-16

或者通过使用 npm 来安装：

```
npm install -g yarn
```

```
C:\Users\think>npm install -g yarn
C:\Users\think\AppData\Roaming\npm\yarnpkg -> C:\Users\think\AppData\Roaming\npm
\node_modules\yarn\bin\yarn.js
C:\Users\think\AppData\Roaming\npm\yarn -> C:\Users\think\AppData\Roaming\npm\no
de_modules\yarn\bin\yarn.js
+ yarn@0.27.5
added 207 packages in 41.379s
```

检验是否安装成功：

```
yarn --version
```

```
C:\Users\think>yarn --version
0.27.5
```

mac 安装：

参考 https://yarnpkg.com/en/docs/install#mac-tab。

1.1.3.3 yarn 常用命令

开始新项目：

```
yarn init
```

添加依赖包：

```
yarn add [package]
yarn add [package]@[version]
yarn add [package]@[tag]
```

升级依赖包：

```
yarn upgrade [package]
yarn upgrade [package]@[version]
yarn upgrade [package]@[tag]
```

移除依赖包：

```
yarn remove [package]
```

安装依赖包：

```
yarn 或者 yarn install
```

查找依赖关系：

```
yarn why
```

全局：

```
yarn global
```

1.1.3.4 yarn 工作流

创建一个新项目；
增加/更新/删除依赖；
安装/重装你的依赖；
和版本控制一起工作；
持续集成。

1.1.4　npm vs yarn

1.1.4.1　yarn.lock 文件

npm 和 yarn 都是用 package.json 来跟踪项目的依赖。npm 使用 npm shrinkwrap 命令来生成一个锁文件，在读取 package.json 文件前会先读取锁文件。npm 需要重新安装。

yarn 使用 yarn.lock 锁文件来保证安装包的版本号，yarn 总会自动更新 yarn.lock。

1.1.4.2　并行安装

npm 安装一个包时，会进行一系列的任务，这些任务的执行是按顺序依次执行的。

yarn 安装一个包时，并行执行这些任务，提高了效率。

1.2　ide 相关配置

1.2.1　常用 ide

sublime text;

Atom;

Webstorm;

Visual Studio Code;

Nodepad++;

Vim 等。

1.2.2　Sublime text

安装插件的方法：

安装 Package Control。

通过 ctrl+` 快捷键或者 View > Show Console 菜单打开控制台，复制粘贴回车如下代码即可。

Sublime text3：

```
import urllib.request,os,hashlib; h = 'df21e130d211cfc94d9b0905775a7c0f'
+ '1e3d39e33b79698005270310898eea76'; pf = 'Package Control.sublime-
package'; ipp = sublime.installed_packages_path(); urllib.request.
install_opener( urllib.request.build_opener( urllib.request.ProxyHandle
r()) ); by = urllib.request.urlopen( 'http://packagecontrol.io/' +
pf.replace(' ', '%20')).read(); dh = hashlib.sha256(by).hexdigest();
print('Error validating download (got %s instead of %s), please try manual
install' % (dh, h)) if dh != h else open(os.path.join( ipp, pf),
'wb' ).write(by)
```

Sublime Text2：

```
import urllib2,os,hashlib; h = 'df21e130d211cfc94d9b0905775a7c0f' +
'1e3d39e33b79698005270310898eea76'; pf = 'Package Control.sublime-
package'; ipp = sublime.installed_packages_path(); os.makedirs( ipp ) if
not os.path.exists(ipp) else None; urllib2.install_opener( urllib2.build
_opener( urllib2.ProxyHandler()) ); by = urllib2.urlopen( 'http://
packagecontrol.io/' + pf.replace(' ', '%20')).read(); dh = hashlib.
sha256(by).hexdigest(); open( os.path.join( ipp, pf), 'wb' ).write(by) if
dh == h else None; print('Error validating download (got %s instead of %s),
please try manual install' % (dh, h) if dh != h else 'Please restart Sublime
Text to finish installation')
```

1.2.2.1 Vue 文件插件

Vue 语法高亮插件：

Sublime text；

安装方法详见 https://github.com/vuejs/vue-syntax-highlight。

Ctrl+Shift+P，输入 install，如图 1-17 所示，选择 Package Control:Install Package：

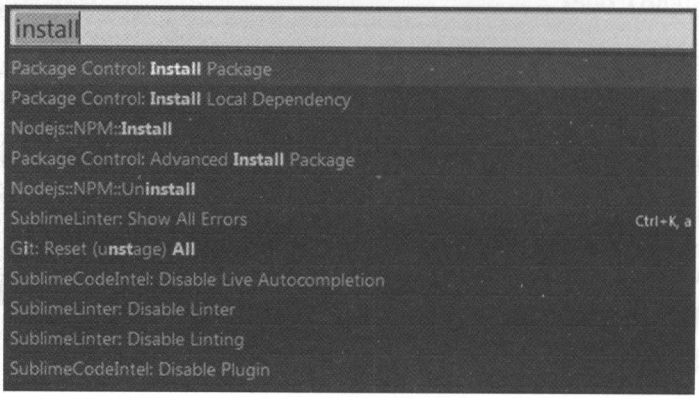


图 1-17

在接下来的输入框中输入 Vue Syntax Highlight 后安装。
安装完成。

1.2.2.2　jsx 语法高亮

Sublime text：
选择 Package Control:Install Package：
输入 babel 安装，如图 1-18 所示。

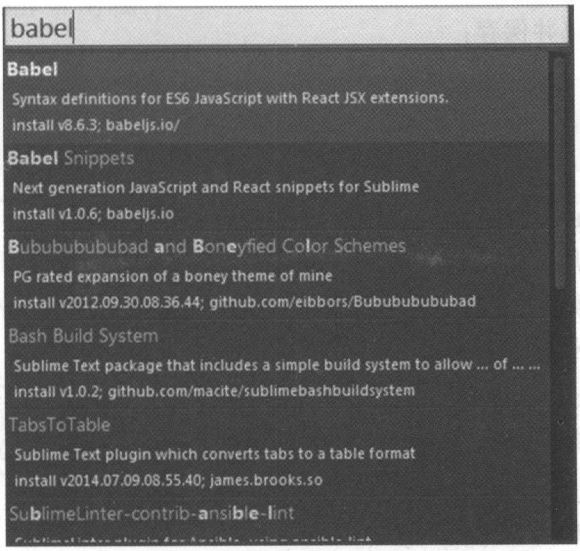
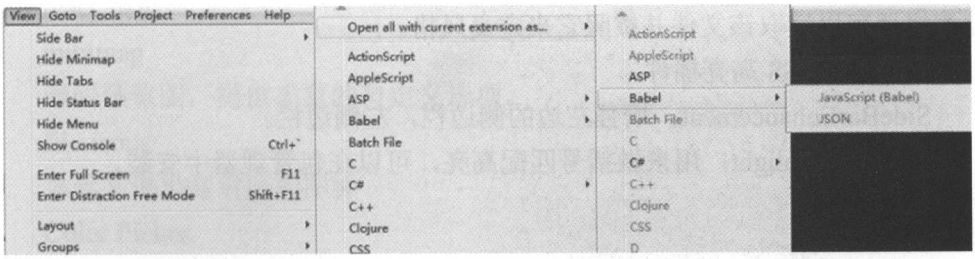


图 1-18

打开.js，.jsx 后缀的文件；
打开菜单 view→Syntax→Open all with current extension as...→Babel→JavaScript (Babel)。



选择 babel 为默认 javascript，打开 syntax 配置完成。

1.2.2.3 其他常用插件

Emmet：编辑器中最流行的插件之一。按下 Tab 键，Emmet 就能把一个缩写展开成一个 HTML 和 CSS 的代码块。

Git：在编辑器中直接和 git 协同工作，不需要在 Sublime 和终端间相互切换。还具有 tag 自动补全功能和快速提交功能。

打开 Package Control 后搜索 git 回车即可，安装完成后打开菜单栏 Preferences/Package Settings/Git/Settings - User。

输入如下代码并保存：

```
{
    "git_command": "git 安装目录\\Git\\cmd\\git.exe"
}
```

AutoFileName：快捷输入文件名，自动补全文件路径。

DocBlockr：用来生成代码注释，是编写代码文档的有效工具。当输入/**并且按下 Tab 键的时候，这个插件会自动解析任何一个函数并且为你准备好合适的模板。

Alignment：快捷键操作，[等号]对齐功能。

SublimeLinter：代码检查的功能，高亮提示用户编写的代码中存在的不规范和错误的写法，支持 JavaScript、CSS、HTML、Java、PHP、Python、Ruby 等十多种开发语言。

ConverToUTF8：文件转码成 UTF-8。

JSFormat：JavaScript 的代码格式化插件。

GitGutter & Modific：实时的 diff 工具，可以高亮显示相对于上次提交所有变动的行。

EditorConfig：帮助开发者在不同的编辑器、IDE 之间定义和维护统一的编程风格。EditorConfig 工程包含一个文件，定义了编程风格，文本编辑器插件集合，让编辑器可以读取该文件并按照它来定义风格。

LESS：LESS 高亮插件。

SideBarEnhancement：增强左边的侧边栏，左侧边栏。

BracketHighlight：用来做括号匹配高亮，可以在包管理器中安装。

1.2.3 Atom

Atom 是 Github 专门为程序员推出的一个跨平台文本编辑器。它很大程度上

集成了 SublimeText 的功能，但又不仅如此。

1.2.3.1　Atom 的插件安装

Ctrl+Shift+P（在 Mac 环境下是 Command+Shift+P）打开设置界面，点击 install，在出来的页面输入框中输入你想要安装的插件名，搜索这个插件，然后在出现的插件选项中点击下载。

1.2.3.2　Atom 常用插件

插件的安装推荐使用 apm

emmet

html 补全

active-power-mode

文字泡沫

power-mode

随地泡沫

autocomplete-plus

当输入的时候，提供可能的候选项。

atom-beautify

格式化代码，统一的编码风格

atomic-chrome

chrome 里面的编辑框直接在 atom 上编辑

atom-html-preview

预览 html 页面

atom-react-preview

react 预览

minimap

源码预览图，提供丰富的自定义选项。

file-icons

显示文件类型对应的图标

Color Picker

在编辑器里面挑选颜色

git-control

git 面板

language-JavaScript-jsx

jsx 扩展

markdown-preview-plus

markdown 预览

Script 在编辑器里运行代码，通过文件名，选中的代码或者行号来运行代码。

linter

代码风格审查。注意，安装 linter 需要安装相关语言代码的风格审查工具才能生效。

tree-view-git-status

文件夹 git 状态

tree-view-background

文件栏背景图

filecolor

文件名颜色

windows-context-menu

给 windows 添加打开快捷方式。

remote-ftp

服务器文件的树状结构显示

browser-plus

浏览器

preview-plus

浏览一切

server-script

同步/运行脚本到服务器

open-in-browser

在浏览器中打开

language-babel

jsx 语法。

1.2.4 WebStorm

WebStorm 是 JetBrains 公司旗下的商业软件产品。是一款前端开发 IDE（集

成开发环境），它是一个智能的，强大的代码编辑器。

WebStorm 本身集成了语法着色，代码补全，代码分析，代码导航，格式化，快捷键，参考文档，实时纠错提示，代码重构，脚本调试，版本控制，单元测试，工程管理，收藏夹，Zen Coding，文件模板，代码片段，语言混编，Git 等版本控制工具，等等。

缺点是占用内存较大。

1.2.5　VSCode

Visual Studio Code（简称 VSCode）是一个轻量级的 Web 集成开发环境 on Linux，Mac and Windows！

常用插件

Auto Close Tag

自动闭合 HTML 标签

Auto Import

Typescript 自动 import 提示

Auto Rename Tag

修改 HTML 标签时，自动修改匹配的标签

Beautify css/sass/scss/less

css/sass/less 格式化

Better Comments

编写更加人性化的注释

Bookmarks

添加行书签

Can I Use

HTML5、CSS3、SVG 的浏览器兼容性检查 Code Runner

运行选中代码段（支持大量语言，包括 Node）

Code Spellchecker

单词拼写检查

CodeBing

在 VSCode 中弹出浏览器并搜索，可编辑搜索引擎

Color Highlight

颜色值在代码中高亮显示

Color Info
小窗口显示颜色值，rgb，hsl，cmyk，hex，等等
Color Picker
拾色器
Document This
注释文档生成
ESLint
ESLint 插件，高亮提示
EditorConfig for VS Code
EditorConfig 插件
Emoji
在代码中输入 emoji
File Peek
根据路径字符串，快速定位到文件
Font-awesome codes for html
FontAwesome 提示代码段
Git Blame
在状态栏显示当前行的 Git 信息
Git History(git log)
查看 git log
GitLens
显示文件最近的 commit 和作者，显示当前行 commit 信息
Guides
高亮缩进基准线
HTML CSS Class Completion
CSS class 提示
HTML CSS Support
css 提示（支持 Vue）
HTMLHint
HTML 格式提示
htmltagwrap
包裹 HTML

Import Cost
行内显示导入（import/require）包的大小。
Indenticator
缩进高亮
JavaScript (ES6) code snippets
ES6 语法代码段 JavaScript Standard Style
Standard 风格 JSON to TS
JSON 结构转化为 typescript 的 interface
JSON Tools
格式化和压缩 JSON
Less IntelliSense
less 变量与混合提示
Lodash
Lodash 代码段
Log Wrapper
生产打印选中变量的代码
Node modules resolve
快速导航到 Node 模块
Code Outline
展示代码结构树
Output Colorizer
彩色输出信息
Path Autocomplete
路径完成提示
Path Intellisense
另一个路径完成提示
PostCss Sorting
css 排序
Prettify JSON
格式化 JSON
Project Manager
快速切换项目

REST Client
发送 REST 风格的 HTTP 请求
React Native Storybooks
storybook
预览插件，支持 react
React Playground
为编辑器提供一个 react 组件运行环境，方便调试。
React Standard Style code snippets
react standar 风格代码块
Sass sass 插件
Sort lines
排序选中行 String Manipulation
字符串转换处理（驼峰、大写开头、下画线，等等）。
TSLint
TypeScript 语法检查
Test Spec Generator
测试用例生成（支持 chai、should、jasmine）
TypeScript Import
TS 自动 import
TypeSearch
TS 声明文件搜索
Version Lens
package.json 文件显示模块当前版本和最新版本
View Node Package
快速打开选中模块的主页和代码仓库
VueHelper
Vue2 代码段（包括 Vue2 API、vue-router2、vuex2）
filesize
状态栏显示当前文件大小
ftp-sync
同步文件到 ftp
gitignore

.gitignore 文件语法
htmltagwrap
快捷包裹 html 标签
language-stylus
Stylus 语法高亮和提示
markdownlint
Markdown 格式提示
npm Intellisense
导入模块时，提示已安装模块名称 npm，运行 npm 命令
stylelint
css/sass/less 代码风格
vetur
目前比较好的 Vue 语法高亮
vscode-database
操作数据库，支持 mysql 和 postgres
vscode-icons
文件图标，方便定位文件
vscode-random
随机字符串生成器
vscode-styled-components
styled-components 高亮支持
vscode-styled-jsx
styled-jsx 高亮支持。

1.3　本章小结

本章主要介绍搭建本地 Node 环境的方法，以及常用的 ide 编辑器的一些插件，旨在为读者进一步了解搭建 Vue 工程提供开发环境的准备。

读 者 圈

第 2 章　从零搭建 Vue 工程

2.1　本地开发需要哪些工具

相信大家在上一章已经完成了 Node 环境的搭建，并对 npm、Node 有了一些较深层次的理解。那么在这一章里，我会一步一步地向大家介绍本地开发 Vue 所需要的一些工具及它们的一些基本知识。

本地构建 Vue 应用主要有以下几种方式，如图 2-1 所示。

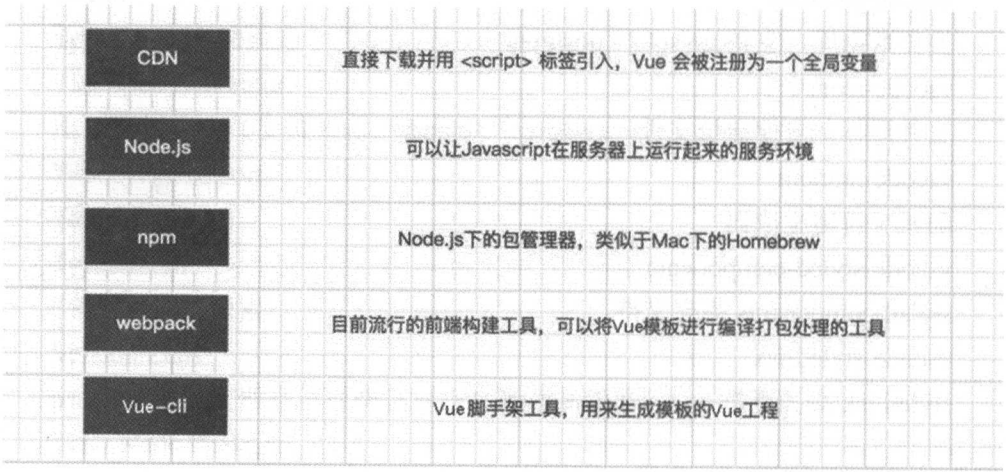


图 2-1

当然，我们可以直接通过 cdn 将 Vue 的源文件引入到你的项目中去。

推荐：unpkg，会保持和 npm 发布的最新版本一致。可以在 unpkg.com/Vue 浏览 npm 包资源。也可以从 jsdelivr 或 cdnjs 获取，不过这两个服务版本更新可能略滞后。

在本章中，我会通过安装 Vue-cli 这个 Vue 脚手架工具，透过这个工具会向大家介绍一些 Nodejs 的服务以及目前最流行的前端构建工具——webpack。

2.2　搭建 Vue 工程

2.2.1　Vue-cli

Vue-cli 是 Vue 官方所提供的一个脚手架工具，这里面默认继承了 express 以及 webpack 打包工具。首先，我们默认大家都已经安装了 Node.js（≥4.x, 6.x preferred），以及 npm 包管理器（当然如果大家觉得 npm 较慢，可以使用国内淘宝的 cnpm）。

2.2.1.1　安装 Vue-cli

```
npm install -g Vue-cli
```

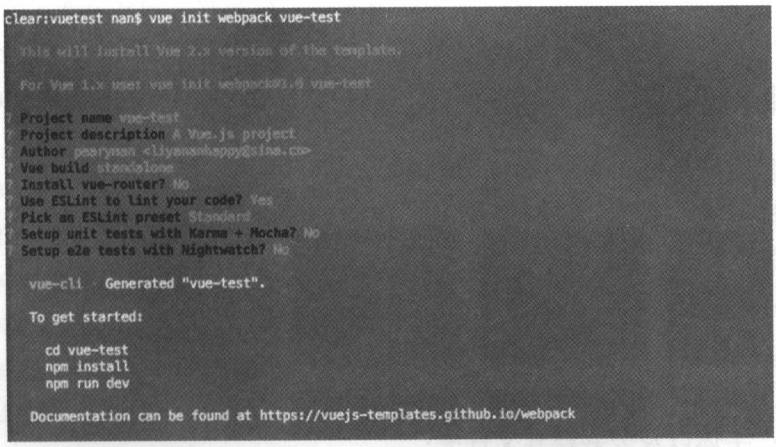
注意：-g 是表示全局安装，不加-g 表示本项目安装。

2.2.1.2　初始化项目

Vue init <template-name> <project-name>

```
Vue init webpack Vue-test
```

初始化的前面是我们需要的构建工具，这里选用 webpack，后面是我们这个项目的名字。当然在这里面会出现一些有关 Vue 的配置选项，大家可以根据自己的使用条件进行选择。比如 Vue-router、ESlint、Karma…我在这里只是演示搭建 Vue 的基本环境，没有加上单元测试等工具。



很明显，在我们初始化完 Vue-test 这个项目后，Vue-cli 已经很人性化地将开始操作告诉我们了。

```
cd Vue-test (进入 Vue-test 文件夹)
  npm install  (将内部的依赖进行安装)
  npm run dev (开始 dev 环境)
```

然后，我们运行 npm run dev 之后，会在浏览器上出现下面的页面，如图 2-2 所示。

图 2-2

这样，我们就利用脚手架工具成功地启动了一个 Vue 的页面。

2.2.2 脚手架项目构成分析

因为我用的是 github 的 atom 编辑器，所以在我们编辑器下所出现的项目目录是这个样子的。下面，我们来看一下在下一级目录上，每一个文件夹以及配置文件的用处，如图 2-3 所示。

build：webpack 的一些配置文件以及服务启动文件；

config：多为 build 中所依赖的文件；

src：页面以及逻辑文件夹；

static：字体以及公共样式文件夹；

图 2-3

.babelrc：es6 编译文件配置，将 es6 编译为 es5；

.editorconfig：编写风格配置文件，比如缩进文件格式，等等；

.eslintignore：忽略检测一些文件格式，比如我们默认忽略检测 build 以及 config 中的 js；

.eslintrc.js：代码规范化配置文件；

.gitignore：忽略上传一些文件或配置；

.postcsssrc.js：用 js 来处理 css；

index.html：主文件入口；

package.json：npm 依赖以及开发生产环境所来的模块包；

README.md：解释说明一下本项目是做什么的。

2.3 webpack 配置

webpack 配置如图 2-4 所示。

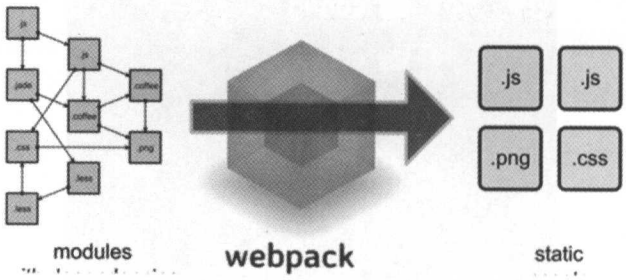


图 2-4

2.3.1 webpack 简介

webpack 是当下最热门的前端资源模块化管理和打包工。它不像 requirejs 那样通过 rjs 进行打包出来的是一个很臃肿的包，它可以将很多松散的模块按照一定的规则打包成符合生产环境的前端资源。同时，它还可以按需加载，等到实际需要的时候进行异步加载。通过 loader 的转换，在项目中任何形式的资源都可以被理解为模块。比如图片、css、less、sass，等等。

同时，随着前端的门槛在逐渐提高，webpack 已经成为大部分前端项目打包工具的首要选择工具。而 grunt、gulp、browserify 等也逐渐沦为辅助工具或者直接被替代。

之前的老式开发方式是你的 html 语言、层叠样式以及 javascript 都是分离的。这也就意味着你必须分别管理它们中的每一个，并且你要要求每一个都达到产品级别。

webpack 识图通过提出一个大胆的想法来减轻开发者的负担，如果一部分的开发过程可以自动处理依赖关系会怎么样？如果我们可以简单地写代码，让构建过程最终只根据需求管理自己会怎么样？

webpack 的工作方式：如果 webpack 了解依赖关系，它会捆绑我们在生产环境中需要的那部分。

换句话说，你根本不需要为了 webpack 来写代码，你只需要关注你的项目代

码，webpack 就会持续地工作。当然，前提是你要配置完成。

简而言之，如果你曾经遇到以下任何一种情况：

- 载入有问题的依赖项；
- 引入了一些你的项目中不需要的 css 以及 js 文件，使你的项目很膨胀；
- 有时候会引入好多次公共库；
- 遇到了作用域问题；
- 需要后端进行配置一些参数，前端才能工作……

这个时候，你或许就应该学习一下 webpack 了。它通过 javascript 轻松处理依赖关系和加载顺序，而不是开发者手动去配置依赖关系和加载顺序。最好的部分是，webpack 甚至可以在服务端运行，这也就意味着可以使用 webpack 做一些构建渐进增强式的网站。

2.3.2　webpack 配置解析

这是当前最火的前端资源构建工具，能将前端资源进行编译、打包、输入、输出等操作。可能之前大家都是在使用 gulp、grunt 之类的构建工具，但是在 Vue-cli 中，我们选择使用了 webpack 这个目前前端最流行的构建工具。我们通过 bulid 目录下面 webpack 的配置，来揭开 webpack 这一层的面纱。

这里主要针对 Build 目录下的 webpack 配置做详细分析。

2.3.2.1　webpack.base.conf.js

1. 主入口文件 entry

```
entry: {
    app: '.src/main.js'
}
```

2. 输出文件 output

```
output: {
    path: config.build.assetsRoot,   // 导出目录的绝对路径
    filename: '[name].js', // 导出文件的文件名
    publicPath: process.env.NODE_ENV === 'production'
      ? config.build.assetsPublicPath
      : config.dev.assetsPublicPath   // 生产模式或开发模式下的 html、js 等文件内部引用的公共路径
  }
```

3. 文件解析 resolve

```
resolve: {
    extensions: ['.js', '.Vue', '.json'], // 自动解析确定的扩展名，使导入模块时不带扩展名
    alias: {  // 创建 import 或 require 的别名
      'Vue$': 'Vue/dist/Vue.esm.js',
      '@': resolve('src'),
      'common': path.resolve(__dirname, '../src/common')
    }
  }
```

4. 模块解析 module

如何处理项目中不同类型的模块。

```
module: {
    rules: [
      {
        test: /\.(js|Vue)$/,    // 对 js、Vue 后缀
        loader: 'eslint-loader', // 进行 eslint 检查
        enforce: "pre", // 编译之前
        include: [resolve('src'), resolve('test')],
        options: {
          formatter: require('eslint-friendly-formatter')
        }
      },
      {
        test: /\.Vue$/, // Vue 文件后缀
        loader: 'Vue-loader', // 使用 Vue-loader 进行处理
        options: VueLoaderConfig // 对 Vue-loader 做额外的选项配置
      },
      {
        test: /\.js$/, //js 文件后缀
        loader: 'babel-loader', // 使用 babel-loader 进行处理
        include: [resolve('src'), resolve('test')] //必须处理包含 src、test 的文件夹
      },
      {
        test: /\.(png|jpe?g|gif|svg)(\?.*)?$/, // 图片后缀
        loader: 'url-loader',  // 使用 url-loader 处理
        query: { // query 是对 loader 做的额外配置
          limit: 10000, // 对于小于 10Kb 的以 base64 进行引用
```

```
          name: utils.assetsPath('img/[name].[hash:7].[ext]')  // 文件
名为name.7位hash值.扩展名
        }
      },
      {
        test: /\.(woff2?|eot|ttf|otf)(\?.*)?$/, // 字体文件
        loader: 'url-loader', // 使用url-loader进行处理
        query: {
          limit: 10000, //字体小于10kb的处理方式
          name: utils.assetsPath('fonts/[name].[hash:7].[ext]') // 文件
名为 name.7位hash值.后缀名
        }
      }
    ]
  }
```

2.3.2.2　webpack.dev.conf.js

```
  module: {
    // 通过传入一些配置来获取rules配置，此处传入了sourceMap: false，表示不生
成sourceMap
    rules: utils.styleLoaders({ sourceMap: config.dev.cssSourceMap })
  }
```

在 util.js 文件下的 styleLoaders 配置如下。

1.　styleLoaders

```
  exports.styleLoaders = function (options) {
    var output = [];  // 定义返回数组，数组中保存的是针对各类型的样式文件的处理方
式

    var loaders = exports.cssLoaders(options) // 调用 cssLoaders 方法返回
各类型的样式对象 (css: loader)
    for (var extension in loaders) {  // 遍历loaders
      var loader = loaders[extension] // 根据遍历获得的key(extension) 来得
到value (loader)
      output.push({
        test: new RegExp('\\.' + extension + '$'),  //  处理文件的类型
        use: loader // 用loader来处理，loader来自loaders【extension】
      })
    }
    return output
  }
```

既然上面的代码调用了 cssloaders，那么，我们就继续看看在 util.js 中的另外一个方法——cssloaders 是怎么实现的吧。

2. cssLoaders

```
exports.cssLoaders = function (options) {
  options = options || {}

  var cssLoader = {
    loader: 'css-loader',
    options: { // options 是 css-loader 的选项配置
      minimize: process.env.NODE_ENV === 'production', // 生成生产环境下的压缩文件
      sourceMap: options.sourceMap // 根据参数是否要生成 sourceMap 文件
    }
  }

  // generate loader string to be used with extract text plugin
  function generateLoaders (loader, loaderOptions) {  // 生成 loader
    var loaders = [cssLoader]  // 默认的 loader 是 css-loader
    if (loader) { // 如果参数 loader 还在
      loaders.push({
        loader: loader + '-loader',
        options: Object.assign({}, loaderOptions, {  // 将 loaderOptions 和 sourceMap 组成一个对象
          sourceMap: options.sourceMap
        })
      })
    }

    // Extract CSS when that option is specified
    // (which is the case during production build)
    if (options.extract) {  // 如果传入的 options 存在 extract 并且为 true
      return ExtractTextPlugin.extract({  // ExtractTextPlugin 分离 js 中引入的 css 文件
        use: loaders, // 处理 loader
        fallback: 'Vue-style-loader' // 没有被提取分离时使用的 loader
      })
    } else {
      return ['Vue-style-loader'].concat(loaders)
    }
  }
```

```
    //
http://Vuejs.github.io/Vue-loader/en/configurations/extract-css.html
    return { // 返回 css 类型对应的 loader 组成的对象  generateLoaders()来生成
loader
      css: generateLoaders(),
      postcss: generateLoaders(),
      less: generateLoaders('less'),
      sass: generateLoaders('sass', { indentedSyntax: true }),
      scss: generateLoaders('sass'),
      stylus: generateLoaders('stylus'),
      styl: generateLoaders('stylus')
    }
  }
```

下面我们再次回到 webpack.dev.conf.js 文件下，介绍本文件下最后一个项目——插件配置。

3. plugins

```
  plugins: [
    new webpack.DefinePlugin({  // 编译时配置的全局变量
      'process.env': config.dev.env // 当前环境为开发环境
    }),
    //
https://github.com/glenjamin/webpack-hot-middleware#installation--usage
    new webpack.HotModuleReplacementPlugin(), // 热更新插件
    new webpack.NoEmitOnErrorsPlugin(), // 不触发错误，即编译后运行的包正常
运行
    // https://github.com/ampedandwired/html-webpack-plugin
    new HtmlWebpackPlugin({ // 自动生成 html 文件，比如编译后文件的引用
      filename: 'index.html', // 生成的文件名
      template: 'index.html', // 模板
      inject: true
    }),
    new FriendlyErrorsPlugin() // 友好的错误提示
  ]
```

以上就是我们的开发环境配置，下面我们来看一下生产环境的配置。

2.3.2.3　webpack.prod.conf.js

生产环境下的 webpack 配置，通过 merge 方法合并 webpack.base.conf.js 基础配置。module 的处理，主要是针对 css 的处理。同样，此处调用了 utils.styleLoaders：

```
    module: {
        rules: utils.styleLoaders({
        sourceMap: config.build.productionSourceMap,
        extract: true
      })
    }
```

1. 输出文件 output

```
    output: {
        path: config.build.assetsRoot,  //导出文件目录
        filename: utils.assetsPath('js/[name].[chunkhash].js'),   //导出的文件名
        chunkFilename: utils.assetsPath('js/[id].[chunkhash].js')   //非入口文件的文件名，而又需要被打包出来的文件命名配置,如按需加载的模块
    }
```

2. 插件 plugins

```
    plugins: [
        // http://Vuejs.github.io/Vue-loader/en/workflow/production.html
        new webpack.DefinePlugin({
          'process.env': env  // 配置全局环境为生产环境
        }),
        new webpack.optimize.UglifyJsPlugin({ // js 压缩插件
          compress: { // 压缩配置
            warnings: false //不显示警告
          },
          sourceMap: true // 生成 sourceMap 文件
        }),
        // extract css into its own file
        new ExtractTextPlugin({ // 将 js 中引入 css 分离插件
          filename: utils.assetsPath('css/[name].[contenthash].css') // 分离出来的 css 文件名
        }),

        new OptimizeCSSPlugin(),    // 压缩提取出的 css,并解决 ExtractTextPlugin 分离出的 js 重复问题(多个文件引用同一个 css 文件)
        // generate dist index.html with correct asset hash for caching.
        // you can customize output by editing /index.html
        // see https://github.com/ampedandwired/html-webpack-plugin
        new HtmlWebpackPlugin({  //生成 html 的插件,引入 css 文件和 js 文件
          filename: config.build.index, // 生成的 html 的文件名
          template: 'index.html', // 依据的模板
```

```
        inject: true, // 注入的 js 文件会被放在 Body 标签中，当值为'head'的时候，
将被放在 head 标签中
        minify: { // 压缩配置
          removeComments: true, // 删除 html 中的注释代码
          collapseWhitespace: true, // 删除 html 中的空白符
          removeAttributeQuotes: true // 删除 html 元素中属性的引号
          // more options:
          //
https://github.com/kangax/html-minifier#options-quick-reference
        },
        // necessary to consistently work with multiple chunks via
CommonsChunkPlugin
        chunksSortMode: 'dependency' //按 dependency 的顺序引入
      }),
      // split vendor js into its own file
      new webpack.optimize.CommonsChunkPlugin({ //分离公共 js 到 vendor 中
        name: 'vendor', // 文件名
        minChunks: function (module, count) { // 声明公告的模块来自
Node_modules 文件夹
          // any required modules inside Node_modules are extracted to
vendor
          return (
            module.resource &&
            /\.js$/.test(module.resource) &&
            module.resource.indexOf(
              path.join(__dirname, '../Node_modules')
            ) === 0
          )
        }
      }),
    //上面虽然已经分离了第三方库，每次修改编译都会改变 vendor 的 hash 值，导致浏览器缓
存失效
    // 原因是 vendor 包含了 webpack 在打包过程中会产生一些运行时代码，运行时代码中实
际上保存了打包后的文件名
    // 当修改业务代码时，业务代码的 js 文件的 hash 值必然会改变。一旦改变必然会导致
vendor 变化。vendor 变化会导致其 hash 值的变化
      // extract webpack runtime and module manifest to its own file in
order to
      // prevent vendor hash from being updated whenever app bundle is
updated
      new webpack.optimize.CommonsChunkPlugin({ //下面主要是将运行时代码提
取到单独的 manifest 文件中，防止其影响 vendor.js
        name: 'manifest',
        chunks: ['vendor']
```

```
    }),
    // copy custom static assets
    new CopyWebpackPlugin([  // 复制静态资源,将static文件内的内容复制到指定
文件夹
      {
        from: path.resolve(__dirname, '../static'),
        to: config.build.assetsSubDirectory,
        ignore: ['.*']  //忽视.*文件
      }
    ])
  ]
```

3. 额外配置

```
  if (config.build.productionGzip) {  // 配置文件开启gzip压缩
    var CompressionWebpackPlugin = require('compression-webpack-plugin')
// 引入压缩文件的组件，该插件会对生成的文件进行压缩，生成一个.gz文件

    webpackConfig.plugins.push(
      new CompressionWebpackPlugin({
        asset: '[path].gz[query]',  // 目标文件名
        algorithm: 'gzip', // 使用gzip压缩
        test: new RegExp( // 满足正值表达式的文件会被压缩
          '\\.(' +
          config.build.productionGzipExtensions.join('|') +
          ')$'
        ),
        threshold: 10240,  // 资源文件大于10KB的时候会被压缩
        minRatio: 0.8  // 最小压缩比达到0.8的时候会被压缩
      })
    )
  }
```

2.3.2.4 npm run dev

```
npm run dev
```

当我们熟悉了上面的配置之后，看看我们执行了 npm run dev 会发生什么。在 package.json 的文件中我们定义了 dev 的运行脚本。

```
"scripts": {
    "dev": "Node build/dev-server.js",
    "build": "Node build/build.js"
  }
```

当运行 npm run dev 的命令时，实际上会运行 dev-server.js 文件。下面我们介绍这个文件的构成情况。

```
require('./check-versions')()  // 检查版本

var config = require('../config')  // Nodejs 环境配置
if (!process.env.NODE_ENV) {
  process.env.NODE_ENV = JSON.parse(config.dev.env.NODE_ENV)
}

var opn = require('opn')  // 强制打开浏览器
var path = require('path')  // 提供文件路径的方法
var express = require('express')  // 借助 express 来启动服务
var webpack = require('webpack')
var proxyMiddleware = require('http-proxy-middleware') // http 代理中间件
var webpackConfig = require('./webpack.dev.conf') // 依赖开发配置

// default port where dev server listens for incoming traffic
var port = process.env.PORT || config.dev.port // 端口号
// automatically open browser, if not set will be false
var autoOpenBrowser = !!config.dev.autoOpenBrowser  // 是否打开浏览器
// Define HTTP proxies to your custom API backend
// https://github.com/chimurai/http-proxy-middleware
var proxyTable = config.dev.proxyTable  // http 的代理 url

var app = express(); // 启动 express
var compiler = webpack(webpackConfig);  // webpack 编译

// devMiddleware 这个中间件是 express 专门为 webpack 开发的中间件
// webpack-dev-middleware 的作用
// 1.将编译后生成的静态文件放在内存中,所以在 npm run dev 后磁盘上不会生成文件
// 2.当文件改变时,会自动编译
// 3.当在编译过程中请求某个资源时，webpack-dev-server 不会让这个请求失败，而是会一直阻塞它，直到 webpack 编译完毕
var devMiddleware = require('webpack-dev-middleware')(compiler, {
  publicPath: webpackConfig.output.publicPath,
  quiet: true
});

// webpack-hot-middleware 的作用就是实现浏览器的无刷新更新,这个中间件是 webpack 中的一个小亮点
var hotMiddleware = require('webpack-hot-middleware')(compiler, {
  log: () => {}
```

```
  });
  // force page reload when html-webpack-plugin template changes
  //声明hotMiddleware无刷新更新的时机:html-webpack-plugin 的template更改之
后
  compiler.plugin('compilation', function (compilation) {
    compilation.plugin('html-webpack-plugin-after-emit', function (data,
cb) {
      hotMiddleware.publish({ action: 'reload' })
      cb()
    });
  });

  // proxy api requests
  //将代理请求的配置应用到express服务上
  Object.keys(proxyTable).forEach(function (context) {
    var options = proxyTable[context]
    if (typeof options === 'string') {
      options = { target: options }
    }
    app.use(proxyMiddleware(options.filter || context, options))
  })

  // handle fallback for HTML5 history API
  //使用connect-history-api-fallback匹配资源
  //如果不匹配，就可以重新定向到指定地址
  app.use(require('connect-history-api-fallback')())

  app.use(devMiddleware)  // 应用devMiddleware中间件
  app.use(hotMiddleware)  // 应用hotMiddleware中间件

  // 配置express静态资源目录
  var   staticPath   =   path.posix.join(config.dev.assetsPublicPath,
config.dev.assetsSubDirectory)
  app.use(staticPath, express.static('./static'))

  var uri = 'http://localhost:' + port

  devMiddleware.waitUntilValid(function () {  //编译成功后打印uri
    console.log('> Listening at ' + uri + '\n')
  })

  module.exports = app.listen(port, function (err) {  // 启动express服
务
    if (err) {
```

```
    console.log(err)
    return
  }

  // when env is testing, don't need open it
  // 满足条件则自动打开浏览器
  if (autoOpenBrowser && process.env.NODE_ENV !== 'testing') {
    opn(uri)
  }
})
```

2.3.2.5　npm run build

由于在 package.json 中的配置，npm run build 执行的是 build.js 文件。

```
require('./check-versions')()

process.env.NODE_ENV = 'production' // 设置当前的环境为production

var ora = require('ora') // 终端显示的转轮 loading
var rm = require('rimraf') // Node 环境下 rm -rf 的命令库
var path = require('path') // 文件路径处理
var chalk = require('chalk') // 终端显示带颜色的文字
var webpack = require('webpack')
var config = require('../config')
var webpackConfig = require('./webpack.prod.conf') // 生产环境下的
webpack 配置

var spinner = ora('building for production...') // 在终端显示 ora 库的
loading 效果
spinner.start()

// 删除已编译文件
rm(path.join(config.build.assetsRoot,
config.build.assetsSubDirectory), err => {
  if (err) throw err
  //在删除完成的回调函数中，开始编译
  webpack(webpackConfig, function (err, stats) {
    spinner.stop() // 停止 loading
    if (err) throw err
    // 在编译完成的回调函数中,在终端输出编译的文件
    process.stdout.write(stats.toString({
      colors: true,
      modules: false,
```

```
        children: false,
        chunks: false,
        chunkModules: false
      }) + '\n\n')

      console.log(chalk.cyan('  Build complete.\n'))
      console.log(chalk.yellow(
        '  Tip: built files are meant to be served over an HTTP server.\n'
+
        '  Opening index.html over file:// won\'t work.\n'
      ))
    })
  })
```

读 者 圈

第3章　API详解

3.1　全局变量

Vue.config 是 Vue 的全局配置对象，包含 Vue 的所有全局配置，可以在启动应用之前修改下列属性。

3.1.1　silent

- 类型：boolean
- 默认值：false
- 用法：

```
Vue.config.silent = true
```

取消所有的日志警告。

3.1.2　optionMergeStrategies

- 类型：{ [key: string]: Function }
- 默认值：{}
- 用法：

optionMergeStrategies 主要用于 mixin 以及 Vue.extend()方法时，对于子组件和父组件如果有相同的属性（option）时的合并策略。

下面先来看看默认的合并策略：

```
var defaultStrat = function (parentVal, childVal) {
    return childVal === undefined
        ? parentVal
        : childVal
}
```

*第一个参数作为父实例，第二个参数作为子实例，合并的策略就是，子组件*的选项不存在时，才会使用父组件的选项；如果子组件的选项存在，使用子组件自身的。

以上所说的都是 Vue 定义的合并策略，当然你也可以自定义某个选项的合并策略。

例如新增一个_my_option 策略：

```
Vue.config.optionMergeStrategies._my_option = function (parent, child,
vm) {
    return child + 1
}
const Profile = Vue.extend({
    _my_option: 1
})
// Profile.options._my_option = 2
```

合并策略选项分别接受第一个参数作为父实例，第二个参数作为子实例，Vue 实例上下文被作为第三个参数传入。

例如，想要修改 watch 的合并策略：

```
Vue.config.optionMergeStrategies.watch = function (toVal, fromVal) {
    // return mergedVal
}
```

3.1.3 devtools

- 类型：boolean
- 默认值：true（产版为 false）
- 用法：
- //务必在加载 Vue 之后，立即同步设置以下内容：

```
Vue.config.devtools  = true
```

配置是否允许 Vue-devtools 检查代码。开发版本默认为 true，生产版本默认为 false。生产版本设为 true 可以启用检查。

devtools 的安装步骤如下。

（1） 找到 Vue-devtools 的 github 项目，并将其 clone 到本地. Vue-devtools：

```
github 下载地址：https://github.com/Vuejs/Vue-devtools
```

（2） 下载好后进入 Vue-devtools-master 工程，安装项目所需要的 npm 包：

```
npm install //如果太慢的话，可以安装一个 cnpm，然后命令换成 cnpm install
```

（3） 编译项目文件：

```
npm run build
```

（4） 修改 manifest.json 中的 persistent 为 true，如图 3-1 所示。

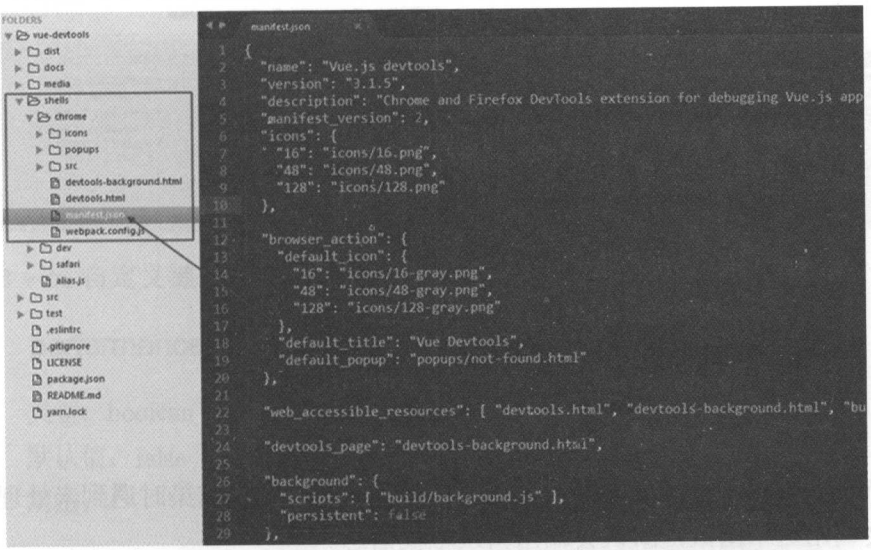


图 3-1

（5） 打开谷歌浏览器设置→扩展程序→勾选开发者模式→添加工程中的 shells→chrome 的内容或者直接拖动 shells→chrome，如图 3-2 所示。

图 3-2

（6） 添加 Vue-devtools 扩展程序后，在调试 Vue 应用时，chrome 开发者工具中会看到 Vue 一栏，点击之后就可以看见当前页面 Vue 对象的一些信息。Vue-devtools 使用起来还是比较简单的，上手非常容易，这里就不细讲其使用说明了，如图 3-3 所示。

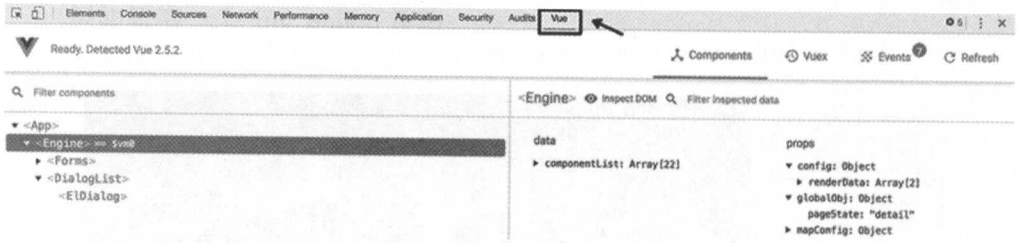


图 3-3

3.1.4 errorHandler

- 类型：Function
- 默认值：默认抛出错误
- 用法：

Vue.config.errorHandler 是一个全局钩子函数。当组件渲染时遇到未处理的异常，会调用这个函数，默认会输出错误堆栈信息。

```
Vue.config.errorHandler = function (err, vm) {
    // handle error
}
```

指定组件的渲染和观察期间未捕获错误的处理函数。这个处理函数被调用时，可获取错误信息和 Vue。

3.1.5 ignoredElements

- 类型：Array<string>
- 默认值：[]
- 用法：

```
• Vue.config.ignoredElements = [
•     'my-custom-web-component', 'another-web-component'
]
```

使 Vue 忽略在 Vue 之外的自定义元素。否则，它会假设你忘记注册全局组件或者拼错了组件名称，从而抛出一个关于 Unknown custom element 的警告。

3.1.6　keyCodes

- 类型：{ [key: string]: number | Array<number> }
- 默认值：{}
- 用法：

```
Vue.config.keyCodes = {
    v: 86,
    f1: 112,
    mediaPlayPause: 179,
    up: [38, 87]
}
```

给 v-on 自定义键位别名。

3.1.7　performance

- 类型：boolean
- 默认值：false（自 2.2.3 起）
- 用法：

```
Vue.config.performance = true
```

设置为 true，以在浏览器开发工具中启用对组件初始化、渲染和打补丁的性能追踪。只适用于开发模式和支持 performance.mark API 的浏览器。

3.1.8　productionTip

- 类型：boolean
- 默认值：true
- 用法：

```
Vue.config.productionTip = false
```

设置为 false，以阻止 Vue 在启动时生成生产提示。

3.2 模板语法

都说 Vue 的学习曲线平缓，使用基于 HTML 的模板语法是一大原因。Vue 允许开发者声明式地将 DOM 绑定至底层 Vue 实例的数据。所有 Vue.js 的模板都是合法的 HTML，所以能被遵循规范的浏览器和 HTML 解析器解析。

模板语法的第一个主要概念是“插值”。数据绑定最常见的形式就是使用“Mustache”语法（双大括号）的文本插值：

```
<span>Message: {{ msg }}</span>
```

Mustache 标签将会被替代为对应数据对象（data）上 msg 属性的值。一旦 msg 属性产生变化，插值处的内容都会更新。

对于所有的数据绑定，Vue.js 都提供了完全的 JavaScript 表达式给予支持。如下的表达式都可以执行：

```
<div>
    <!-- 一元运算符 -->
    <p>hello {{world + '!'}}</p>
    <!-- 模板字符串 -->
    <p>{{`hello ${world}!`}}</p>
    <!-- 三元运算符 -->
    <p>{{ ok ? 'YES' : 'NO' }}</p>
    <!-- 字符串操作 -->
    <p>{{ message.split('').reverse().join('') }}</p>
</div>
```

这些表达式会在所属 Vue 实例的数据作用域下作为 JavaScript 被解析。有个限制就是，每个绑定都只能包含单个表达式，所以下面的例子都不会生效。

```
<div>
    <!-- 这是语句，不是表达式 -->
    <p>{{ var a = 1 }}</p>
    <!-- 流控制也不会生效，请使用三元表达式 -->
    <p>{{ if (ok) { return message } }}</p>
</div>
```

3.3 指令

模板语法的第二个主要概念是指令。指令（Directives）是带有 v- 前缀的特

殊属性。

指令属性的值预期是单个 JavaScript 表达式（v-for 是例外），而且必须添加到一个 Vue 模板的元素上才能生效。

它的职责是当表达式的值发生改变时，将其产生的连带影响，响应式地作用于 DOM。

下面我们详细讲解 Vue 提供的各种指令。

3.3.1　v-text

v-text 主要用来更新元素 textContent，可以类比 JS 的 text 属性。

```
<span v-text="msg"></span>
<!-- 和下面的一样 -->
<span>{{ msg }}</span>
```

如果想改变部分 textContent，则必须使用之前讲过的文本插值能力。

3.3.2　v-html

文本插值的方式（双大括号）会将数据解释为纯文本，而非 HTML。为了输出真正的 HTML，你需要使用 v-html 指令，可以类比 JS 的 innerHtml 属性：

```
<div v-html="rawHtml"></div>
```

这个 div 的内容将会被替换成属性值 rawHtml，直接作为 HTML 进行渲染。

值得注意的是，在网站上动态渲染任意 HTML 是非常危险的，因为容易导致 XSS 攻击。关于 XSS 攻击的详细原理和防御方式，不属于本书讲解的内容，有兴趣的读者可以自行查阅相关资料

3.3.3　v-pre

v-pre 主要用来跳过这个元素和它的子元素编译过程。可以用来显示原始的 Mustache 标签。跳过大量没有指令的节点加快编译，例如：

```
<div id="app">
  <span v-pre>{{message}}</span>
  <span>{{message}}</span>
</div>

<script type="text/javascript">
```

```
    var app=new Vue({
     el:'#app',
     data:{
      message:'hello world'
     }
    });
</script>
```

第一个 span 里的内容不会被编译，显示为{{message}}，第二个 span 里的内容会被编译，显示为 hello world。

3.3.4 v-cloak

v-cloak 这个指令是用来保持在元素上直到关联实例结束时进行编译。和 CSS 规则如[v-cloak] {display: none}一起用时，这个指令可以隐藏未编译的 Mustache 标签，直到实例准备完毕。例如：

```
<div id="app">
    <div>
        {{ message }}
    </div>
</div>

<script type="text/javascript">
    new Vue({
        el: '#app',
        data: {
            message: 'hello world'
        }
    })
</script>
```

绑定 Vue 实例，在页面加载时会闪烁，先显示：

```
<div>
    {{ message }}
</div>
```

然后才会编译为：

```
<div>
    hello world
</div>
```

这时候我们使用 v-cloak 可以解决这一问题：

```
<style>
    [v-cloak] {
        display: none;
    }
</style>

<div id="app" v-cloak>
    <div>
        {{ message }}
    </div>
</div>

<script type="text/javascript">
    new Vue({
        el: '#app',
        data: {
            message: 'hello world'
        }
    })
</script>
```

这个 div 在页面渲染的时候不会显示，直到编译结束。

3.3.5　v-once

v-once 关联的实例，只会渲染一次。之后的重新渲染，实例及其所有的子节点将被视为静态内容并跳过，这可以用于优化更新性能。

```
<!-- 单个元素 -->
<span v-once>This will never change: {{msg}}</span>
<!-- 有子元素 -->
<div v-once>
    <h1>comment</h1>
    <p>{{msg}}</p>
</div>
<!-- 组件 -->
<my-component v-once :comment="msg"></my-component>
<!-- `v-for` 指令-->
<ul>
    <li v-for="i in list" v-once>{{i}}</li>
</ul>
```

在以上的这些例子中，msg、list 即使产生变化，也不会被重新渲染。

3.3.6 v-if

在早期的前端开发中，我们只能在 JS 代码里面做条件判断，而在 Vue.js 中，我们使用 v-if 指令来实现条件渲染，Vue 会根据表达式的值的真假条件来渲染元素。在切换时元素及它的数据绑定，组件被销毁并重建。例如：

```
<a v-if="ok">Yes</a>
```

如果属性值 ok 为 true，则显示：

```
<a>Yes</a>
```

否则不会渲染这个元素。

因为 v-if 是一个指令，需要将它添加到一个元素上。但是如果我们想切换多个元素呢？此时我们可以把一个<template>元素当作包装元素，并在上面使用 v-if。最终的渲染结果不会包含<template>元素。

```
<template v-if="ok">
    <h1>Title</h1>
    <p>Paragraph 1</p>
    <p>Paragraph 2</p>
</template>
```

如果属性值 ok 为 true，则显示：

```
<h1>Title</h1>
<p>Paragraph 1</p>
<p>Paragraph 2</p>
```

3.3.7 v-else

有过任何语言编程经验的读者都知道，else 是搭配 if 使用的，开发者可以使用 v-else 指令来表示 v-if 的“else 块”：

```
<a v-if="ok">Yes</a>
<a v-else>No</a>
```

v-else 元素必须紧跟在 v-if 或者 v-else-if 元素的后面，否则它将不会被识别。

3.3.8　v-else-if

如果我们想要用 if/else 来实现 switch 语句（基于不同条件，执行不同的代码块），我们必须使用金字塔结构的 if 嵌套，这种书写方式，明显在代码可读性上有很大问题。Vue 提供了 v-else-if，充当 v-if 的“else-if 块”。可以链式地使用多次：

```
<div v-if="type === 'A'">
    A
</div>
<div v-else-if="type === 'B'">
    B
</div>
<div v-else-if="type === 'C'">
    C
</div>
<div v-else>
  Not A/B/C
</div>
```

类似于 v-else、v-else-if，必须紧跟在 v-if 或者 v-else-if 元素之后。

3.3.9　v-show

另一个用于根据条件展示元素的选项是 v-show 指令：

```
<h1 v-show="ok">Hello!</h1>
```

不同的是带有 v-show 的元素始终会被渲染并保留在 DOM 中。v-show 是简单地切换元素的 CSS 属性 display。注意，v-show 不支持<template>语法，也不支持 v-else。

v-if 是“真正的”条件渲染，因为它会确保在切换过程中，条件块内的事件监听器和子组件适当地被销毁和重建。v-if 也是惰性的：如果在初始渲染时条件为假，则什么也不做，直到条件第一次变为真时，才会开始渲染条件块。相比之下，v-show 就简单得多，不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。一般来说，v-if 有更高的切换开销，而 v-show 有更高的初始渲染开销。因此，如果需要非常频繁地切换，则使用 v-show 较好；如果在运行时条件不太可能改变，则使用 v-if 较好。

3.3.10 v-for

我们用 v-for 指令根据遍历数组来进行渲染。之前说过，v-for 指令的值不能用 JS 表达式，而需要以“item in items”形式的特殊语法，items 是源数据组，item 是数组元素迭代的别名。

3.3.10.1 基本用法

```
<ul id="app">
    <li v-for="item in items">
        {{ item.text }}
    </li>
</ul>
<script type="text/javascript">
    var example1 = new Vue({
        el: '#app',
        data: {
            items: [
                {text: 'text1' },
                {text: 'text2' }
            ]
        }
    })
</script>
```

会渲染为：

```
<ul id="app">
    <li>text1</li>
    <li>text2</li>
</ul>
```

3.3.10.2 对父作用域的访问

在 v-for 块中，我们拥有对父作用域属性的完全访问权限。v-for 还支持一个可选的第二个参数为当前项的索引。

```
<ul id="app">
    <li v-for="(item, index) in items">
        {{ parentMessage }} - {{ index }} - {{ item.text }}
    </li>
```

```
</ul>
<script type="text/javascript">
    var example2 = new Vue({
        el: '#app',
        data: {
            parentMessage: 'parent',
            items: [
                { text: 'text1' },
                { text: 'text2' }
            ]
        }
    })
</script>
```

以上代码会渲染为：

```
<ul id="app">
    <li>
        parent - 0 - text1
    </li>
    <li>
        parent - 1 - text2
    </li>
</ul>
```

3.3.10.3　替换分隔符

你也可以用 of 替代 in 作为分隔符：

```
<div v-for="item of items"></div>
```

3.3.10.4　取整数

v-for 也可以取整数，在这种情况下，它将重复多次模板：

```
<div>
  <span v-for="n in 10">{{ n }} </span>
</div>
```

这段代码会渲染为：

```
<div>
  <span>0</span>
  <span>1</span>
```

```
    <span>2</span>
    <span>3</span>
    <span>4</span>
    <span>5</span>
    <span>6</span>
    <span>7</span>
    <span>8</span>
    <span>9</span>
    <span>10</span>
</div>
```

3.3.10.5 <template>应用

当然，v-for 也可以用在<template>中：

```
<ul id="app">
    <template v-for="item in items">
        <li>{{ item.text }}</li>
    </template>
</ul>
<script type="text/javascript">
    var example1 = new Vue({
        el: '#app',
        data: {
            items: [
                {text: 'text1' },
                {text: 'text2' }
            ]
        }
    })
</script>
```

这部分代码会渲染为：

```
<ul id="app">
    <li>text1</li>
    <li>text2</li>
</ul>
```

3.3.10.6 对象属性遍历

你也可以用 v-for 通过一个对象的属性来迭代：

```
<ul id="app" class="demo">
    <li v-for="value in object">
        {{ value }}
    </li>
</ul>

<script type="text/javascript">
new Vue({
    el: '#app',
    data: {
        object: {
            text: 'text1',
            number: 1
        }
    }
})
</script>
```

这部分代码会渲染为：

```
<ul id="app">
    <li>text1</li>
    <li>1</li>
</ul>
```

你也可以提供第二个参数为键名：

```
<div v-for="(value, key) in object">
    {{ key }}: {{ value }}
</div>

<script type="text/javascript">
new Vue({
    el: '#app',
    data: {
        object: {
            text: 'text1',
            number: 1
        }
    }
})
```

这部分代码会渲染为：

```
<ul id="app">
    <li>text: text1</li>
    <li>number: 1</li>
</ul>
```

第三个参数为索引：

```
<div v-for="(value, key, index) in object">
  {{ index }}. {{ key }}: {{ value }}
</div>

<script type="text/javascript">
new Vue({
    el: '#app',
    data: {
        object: {
            text: 'text1',
            number: 1
        }
    }
})
```

这部分代码会渲染为：

```
<ul id="app">
    <li>0. text: text1</li>
    <li>1. number: 1</li>
</ul>
```

3.3.10.7 key

为了提高 Vue 更新 DOM 的性能，你需要为每项提供一个唯一的 key 属性，有相同父元素的子元素必须有独特的 key，重复的 key 会造成渲染错误。

它的工作方式类似于一个属性，所以你需要用 v-bind 来绑定动态值（关于 v-bind，会在稍后的章节中讲到）：

```
<div v-for="item in items" :key="item.id">
  <!-- 内容 -->
</div>
```

Vue 建议尽可能使用 v-for 来提供 key，除非 DOM 内容遍历起来非常简单，或者你是有意识地要依赖于默认行为以便获得性能提升。

3.3.10.8　用 key 管理可复用的元素

Vue 尽可能高效地渲染元素，通常会复用已有元素而不是从头开始渲染。这么做，除了使 Vue 变得非常快之外，还有一些好处。例如，如果你允许用户在不同的登录方式之间切换：

```
<template v-if="loginType === 'username'">
   <label>Username</label>
   <input placeholder="Enter your username">
</template>
<template v-else>
   <label>Email</label>
   <input placeholder="Enter your email address">
</template>
```

那么在上面的代码中切换 loginType 将不会清除用户已经输入的内容。因为两个模板使用了相同的元素，<input>不会被替换掉——仅仅是替换了它的 placeholder。

这样也不总是符合实际需求，所以 Vue 为你提供了一种方式来声明“这两个元素是完全独立的，不要复用它们”。只需添加一个具有唯一值的 key 属性即可：

```
<template v-if="loginType === 'username'">
  <label>Username</label>
  <input placeholder="Enter your username" key="username-input">
</template>
<template v-else>
  <label>Email</label>
  <input placeholder="Enter your email address" key="email-input">
</template>
```

现在，每次切换时，输入框都将被重新渲染。

3.3.10.9　数组更新检测

1. 变异方法

数组的变异方法，指的是让原始数组产生变化的方法。因为原始数组有了变化，所以也会触发 DOM 更新。这些方法如下：

```
push()
pop()
shift()
```

```
unshift()
splice()
sort()
reverse()
```

2. 重塑数组

filter()，concat()和 slice()这些方法不会改变原始数组，但总是返回一个新数组。当使用非变异方法时，可以用新数组替换旧数组：

```
arrA = arrA.concat(arrB);
```

虽然数组直接进行了替换，但是，Vue 为了使得 DOM 元素得到最大范围的重用而实现了一些智能的、启发式的方法，所以用一个含有相同元素的数组去替换原来的数组是非常高效地操作，并不会重新渲染整个列表，这部分可以参考 Virtual DOM 原理部分。

3.3.10.10 注意事项

由于 JavaScript 的限制，Vue 不能检测以下变动的数组。

（1） 当你利用索引直接设置一个项时，例如：

```
arr1[indexOfItem] = newValue
```

（2） 当你修改数组的长度时，例如：

```
arr2.length = newLength
```

为了解决第一类问题，以下两种方式都可以实现：

```
// Vue.set
Vue.set(arr1, indexOfItem, newValue)
// Array.prototype.splice
arr1.splice(indexOfItem, 1, newValue)
```

为了解决第二类问题，你可以使用 splice：

```
arr2.splice(newLength)
```

3.3.10.11 对象更改检测注意事项

还是由于 JavaScript 的限制，Vue 不能检测对象属性的添加或删除：

```
// a 的变化可以检测到
var vm = new Vue({
    data: {
        a: 1
    }
})
// b 的变化检测不到
vm.b = 2
```

对于已经创建的实例，Vue 不能动态添加根级别的响应式属性。但是，可以使用 Vue.set(object, key, value)方法向嵌套对象添加响应式属性。例如，对于：

```
var vm = new Vue({
    data: {
        message: {
            text1: 'hello'
        }
    }
})
```

你可以添加一个新的 text2 属性到 message 对象：

```
Vue.set(vm.message, 'text2', 'world')
```

你还可以使用 vm.$set 实例方法，它只是全局 Vue.set 的别名：

```
this.$set(this.message, 'text2', 'world')
```

有时你可能需要为已有对象赋予多个新属性，比如使用 Object.assign()或_.extend()。在这种情况下，你应该用两个对象的属性创建一个新的对象。所以，如果你想添加新的响应式属性，可以这么做：

```
this.message = Object.assign({}, this.message, {
    text2: 'world',
    text3: '!'
})
```

3.3.10.12　v-for with v-if

当它们处于同一节点，v-for 的优先级比 v-if 更高，这意味着 v-if 将分别重复运行于每个 v-for 循环中。当你想为仅有的一些项渲染节点时，这种优先级的机制会十分有用，例如：

```
<li v-for="todo in todos" v-if="!todo.isComplete">
```

```
        {{ todo }}
    </li>
```

上面的代码只传递了未 complete 的 todos。而如果你的目的是有条件地跳过循环的执行，那么可以将 v-if 置于外层元素（或<template>）上。例如：

```
<ul v-if="todos.length">
    <li v-for="todo in todos">
        {{ todo }}
    </li>
</ul>
<p v-else>No todos left!</p>
```

3.3.11 v-bind

v-bind 主要是用来动态地绑定一个或多个特性。没有参数时，可以绑定到一个包含键值对的对象。注意，此时 class 和 style 绑定不支持数组和对象。

我们来看一个简单的例子，在页面中有一个链接，需要定义 a 标签的 href 属性，我们可以在元素属性里定义：

```
<div id="app">
    <a href="https://waimai.baidu.com"></a>
</div>

<script type="text/javascript">
    var app = new Vue({
        el: '#app'
    });
</script>
```

实际情况中，我们很多时候是从数据中动态获取相应的值，这个时候就可以这么使用：

```
<div id="app">
    <a v-bind:href="url"></a>
</div>

<script type="text/javascript">
    var app = new Vue({
        el: '#app',
        data: {
            url: 'https://waimai.baidu.com'
```

```
        }
    });
</script>
```

当然，我们可以同时绑定多个属性：

```
<div id="app">
    <a v-bind:href="url" v-bind:target="target"></a>
</div>

<script type="text/javascript">
    var app = new Vue({
        el: '#app',
        data: {
            url: 'https://waimai.baidu.com',
            target: '_blank'
        }
    });
</script>
```

v-bind 也可以简写为：

```
<div id="app">
    <a :href="url" :target="target"></a>
</div>
```

3.3.11.1　对象语法

我们可以传给 v-bind:class 一个对象，以动态地切换 class：

```
<body>
    <div id="app">
        <div :class="{ 'is-active': isActive }">test</div>
    </div>

    <!-- ... ... -->

    <script type="text/javascript">
        var app = new Vue({
            el: '#app',
            data: {
                isActive: true
            }
        });
```

```
    </script>
</body>
```

我们可以通过修改 isActive 的值，来修改 class。

也可以在对象中传入更多属性用来动态切换多个 class。此外，v-bind 指令可以和普通的 class 属性共存，例如：

```
<body>
    <div id="app">
        <div   v-bind:class="{   active:   isActive,   'text-danger':
hasError}" >test</div>
    </div>

    <script type="text/javascript">
        var app = new Vue({
            el: '#app',
          data: {
              isActive: true,
              hasError: false
          }
        });
    </script>
</body>
```

你也可以直接绑定数据里的一个对象：

```
<body>
    <div id="app">
        <div v-bind:class="classObject">test</div>
    </div>

    <script type="text/javascript">
        var app = new Vue({
            el: '#app',
          data: {
              classObject: {
                  active: true,
                  'text-danger': false
              }
          }
        });
    </script>
</body>
```

结果和上面是一样的。

v-bind 也可以用于绑定样式，使用行内样式时，关键字是 style，跟在 html 里面直接写行内样式类似。注意样式的写法跟 css 会有些许不同，要使用正确的写法：

```
<div id="app">
    <div v-bind:style="{ color: 'red', fontSize: '12px' }">test</div>
</div>
```

当然，也可以把数据写到 data 里，实现动态绑定：

```
<div id="app">
    <div  :style="{  color:  activeColor,  fontSize:  fontSize  +
'px' }">test</div>
</div>

<script type="text/javascript">
    var app = new Vue({
        el: '#app',
        data: {
            activeColor: 'red',
            fontSize: 12
        }
    });
</script>
```

直接绑定到一个样式对象通常更好，让模板更清晰：

```
<div id="app">
    <div :style="styleObject">test</div>
</div>

<script type="text/javascript">
    var app = new Vue({
        el: '#app',
        data: {
            styleObject: {
                color: 'red',
                fontSize: '12px'
            }
        }
    });
</script>
```

3.3.11.2 数组语法

我们可以把一个数组传给 v-bind:class，可以生成一个 class 如下：

```
<div id="app">
    <div v-bind:class="[activeClass, errorClass]">test</div>
</div>

<script type="text/javascript">
    var app = new Vue({
        el: '#app',
    data: {
            activeClass: 'active',
            errorClass: 'text-danger'
    }
    });
</script>
```

渲染为：

```
<div class="active text-danger"></div>
```

如果你也想根据条件切换 class，可以用三元表达式：

```
<div id="app">
    <div v-bind:class="[isActive ? activeClass : '', errorClass]">
</div>

<script type="text/javascript">
    var app = new Vue({
        el: '#app',
    data: {
            activeClass: 'active',
            errorClass: 'text-danger'
    }
    });
</script>
```

此例始终添加 errorClass，但是，只有在 isActive 是 true 时才能添加 activeClass。

不过，当有多个条件 class 时，这样写有些烦琐，可以在数组语法中使用对象语法：

```
<div id="app">
    <div v-bind:class="[{ active: isActive }, errorClass]">
```

```
</div>

<script type="text/javascript">
    var app = new Vue({
        el: '#app',
    data: {
            activeClass: 'active',
            errorClass: 'text-danger'
    }
    });
</script>
```

v-bind:style 的数组语法可以将多个样式对象应用到一个元素上：

```
<div id="app">
    <div v-bind:style="[baseStyles, overridingStyles]">
</div>

<script type="text/javascript">
    var app = new Vue({
        el: '#app',
    data: {
            fontSize: '12px',
            color: 'red'
    }
    });
</script>
```

3.3.11.3　自动添加前缀

当 v-bind:style 使用需要特定前缀的 CSS 属性时，如 transform，Vue.js 会自动侦测并添加相应的前缀。

3.3.11.4　多重值

开发者可以为 style 绑定属性，提供一个包含多个值的数组，常用于提供多个带前缀的值：

```
<div :style="{ display: ['-webkit-box', '-ms-flexbox', 'flex'] }">
```

这会渲染数组中最后一个被浏览器支持的值。在这个例子中，如果浏览器支持不带浏览器前缀的 flexbox，那么渲染结果会是 display: flex。

3.3.12 v-model

你可以用 v-model 指令在表单控件元素上创建双向数据绑定。它会根据控件类型自动选取正确的方法来更新元素。

v-model 会忽略所有表单元素的 value、checked、selected 特性的初始值。因为它会选择 Vue 实例数据来作为具体的值。你应该通过 JavaScript 在组件 data 的选项中声明初始值。

对于要求 IME（如中文、日语、韩语等，IME 意为‘输入法’）的语言，你会发现 v-model 不会在 ime 输入中得到更新。如果你也想实现更新，请使用 input 事件。

3.3.12.1 input 绑定

```
<div id="app">
    <input v-model="somebody">
    <p>hello {{ somebody }}</p>
</div>

<script type="text/javascript">
    var app = new Vue({
        el: '#app',
    data: {
            somebody: ''
    }
    });
</script>
```

在这个例子中，只要我们在<input>里输入内容，<p>标签里的 text 都会根据输入的内容实时渲染。

3.3.12.2 textarea 绑定

```
<div id="app">
    <span>hello</span>
    <p style="white-space: pre-line">{{ message }}</p>
    <textarea v-model="sombodys"></textarea>
</div>

<script type="text/javascript">
```

```
    var app = new Vue({
        el: '#app',
    data: {
            somebodys: ''
    }
    });
</script>
```

在这个例子中，我们在<textarea>里输入内容，<p>标签里会根据输入的多行文本，渲染多行文字。

3.3.12.3　checkbox 绑定

单个勾选框：

```
<div id="app">
    <input type="checkbox" id="checkbox" v-model="checked">
    <label for="checkbox">{{ checked }}</label>
</div>

<script type="text/javascript">
    var app = new Vue({
        el: '#app',
    data: {
            checked: false
    }
    });
</script>
```

这里绑定的 checked 是一个 Boolean 的类型变量。

而有多个勾选框时，绑定的 checked 变量需要为同一个数组：

```
<div id="app">
    <input type="checkbox" id="checkbox1" value="1" v-model="checked">
    <label for="checkbox1">checkbox1</label>
    <input type="checkbox" id="checkbox2" value="2" v-model="checked">
    <label for="checkbox2">checkbox2</label>
    <input type="checkbox" id="checkbox3" value="3" v-model="checked">
    <label for="checkbox3">checkbox3</label>
    <br>
    <span>Checked Box: {{ checked }}</span>
</div>

<script type="text/javascript">
```

```
    var app = new Vue({
        el: '#app',
    data: {
            checked: []
    }
    });
</script>
```

3.3.12.4 radio 绑定

```
<div id="app">
    <input type="radio" id="radio1" value="1" v-model="picked">
    <label for="one">radio1</label>
    <input type="radio" id="radio2" value="2" v-model="picked">
    <label for="two">radio2</label>
    <span>Picked: {{ picked }}</span>
</div>

<script type="text/javascript">
new Vue({
    el: '#app',
    data: {
        picked: ''
    }
})
</script>
```

3.3.12.5 select 绑定

单选的 select 可以像下面这样绑定到一个 String 变量上：

```
<div id="app">
    <select v-model="selected">
        <option>A</option>
        <option>B</option>
        <option>C</option>
    </select>
    <span>Selected: {{ selected }}</span>
</div>

<script type="text/javascript">
new Vue({
    el: '#app',
```

```
    data: {
        selected: ''
    }
})
</script>
```

多选的 select 则需要绑定到一个数组上：

```
<div id="app">
    <select v-model="selected" multiple>
        <option>A</option>
        <option>B</option>
        <option>C</option>
    </select>
    <span>Selected: {{ selected }}</span>
</div>

<script type="text/javascript">
new Vue({
    el: '#app',
    data: {
        selected: []
    }
})
</script>
```

动态选项，用 v-for 渲染，value 的值也可以动态绑定，其他的表单元素同理：

```
<div id="app">
    <select v-model="selected">
        <option v-for="option in options" v-bind:value= "option.value">
            {{ option.text }}
        </option>
    </select>
    <span>Selected: {{ selected }}</span>
</div>

<script type="text/javascript">
new Vue({
    el: '#app',
    data: {
        selected: '',
        options: [{
            text: 'One',
            value: 'A'
```

```
        }, {
            text: 'Two',
            value: 'B'

        }, {
            text: 'Three',
            value: 'C'

        } ]
    }
})
</script>
```

3.3.12.6 修饰符

1. .lazy

在默认情况下，v-model 在 input 事件中，同步输入框的值与数据（除了上述 IME 部分），但你可以添加一个修饰符 lazy，从而转变为在 change 事件中同步：

```
<!-- 在 "change" 而不是 "input" 事件中更新 -->
<input v-model.lazy="msg" >
```

2. .number

如果想自动将用户的输入值转为 number 类型（如果原值的转换结果为 NaN，则返回原值），可以添加一个修饰符 number 给 v-model 来处理输入值：

```
<input v-model.number="age" type="number">
```

3. .trim

如果要自动过滤用户输入的首尾空格，可以添加 trim 修饰符到 v-model 上过滤输入：

```
<input v-model.trim="msg">
```

3.3.13 v-on

v-on 主要用来监听 DOM 事件，以便执行一些代码块。表达式可以是一个方法名，例如：

```
<div id="app">
    <button v-on:click="consoleLog"></button>
</div>

<script type="text/javascript">
    new Vue({
        el: '#app',
        methods: {
            consoleLog: function(event) {
                console.log(1);
            }
        }
    })
</script>
```

也可以是一个内敛语句：

```
<div id="app">
    <button v-on:click="consoleLog('test')"></button>
</div>

<script type="text/javascript">
    new Vue({
        el: '#app',
        methods: {
            consoleLog: function(res) {
                console.log(res);
            }
        }
    })
</script>
```

如果使用内联语句，语句可以访问一个 $event 属性：

```
<button v-on:click="consoleLog('test', $event)"></button>
```

当然，v-on 也可以缩写为：

```
<button @click="consoleLog('test', $event)"></button>
```

3.3.13.1　事件修饰符

在事件调用过程中，event.preventDefault()或 event.stopPropagation()是非常常

见的需求。Vue 为 v-on 提供了事件修饰符，通过表示的指令后缀来调用修饰符：

```
.stop
.prevent
.capture
.self
.once
<!-- 阻止单击事件冒泡 -->
<a @click.stop="doThis"></a>
<!-- 提交事件不再重载页面 -->
<form @submit.prevent="onSubmit"></form>
<!-- 修饰符可以串联  -->
<a @click.stop.prevent="doThat"></a>
<!-- 只有修饰符 -->
<form @submit.prevent></form>
<!-- 添加事件侦听器时使用事件捕获模式 -->
<div @click.capture="doThis">...</div>
<!-- 只当事件在该元素本身（比如不是子元素）触发时触发回调 -->
<div @click.self="doThat">...</div>
<!-- 点击事件将只会触发一次 -->
<a @click.once="doThis"></a>
```

不像其他只能对原生的 DOM 起作用的修饰符，.once 还能被用到自定义的组件事件上。

3.3.13.2 键值修饰符

Vue 允许为 v-on 在监听键盘事件时，添加关键修饰符：

```
<!-- 只有在 keyCode 是 13 时调用 vm.submit() -->
<input @keyup.13="submit">
```

记住所有的 keyCode 比较困难，所以 Vue 为最常用的按键提供了别名：

```
<input @keyup.enter="submit">
```

全部的按键别名：

```
.enter
.tab
.delete
.esc
.space
```

```
.up
.down
.left
.right
.ctrl
.alt
.shift
.meta
<!-- 以下为鼠标按键修饰符 -->
.left
.right
.middle
```

可以通过全局 config.keyCodes 对象自定义键值修饰符别名：

```
// 可以使用 v-on:keyup.f1
Vue.config.keyCodes.f1 = 112
```

3.3.14　自定义指令

除了以上所有的指令外，Vue 也允许注册自定义指令。很多情况下，我们仍然要对 DOM 元素进行操作，这时候就可以用到自定义指令。

3.3.14.1　钩子函数

指令定义函数提供了如下几个钩子函数。

- bind：只调用一次，指令第一次绑定到元素时调用，用这个钩子函数可以定义一个在绑定时执行一次的初始化动作。
- inserted：被绑定元素插入父节点时调用（父节点存在即可调用）。
- update：所在元素节点更新时调用，可能发生在子节点更新之前。
- componentUpdated：所在元素节点及其子节点更新完成后调用。
- unbind：只调用一次，指令与元素解绑时调用。

每个钩子函数都有以下参数：

- el：绑定的元素，可以用来直接操作 DOM。
- binding：一个对象，包含以下属性。
  - name：指令名，不包括 v- 前缀。
  - value：指令的绑定值，例如：v-my-directive="1 + 1"，value 的值是 2。
  - oldValue：指令绑定的前一个值，仅在 update 和 componentUpdated 钩子

中可用，无论值是否改变都可用。

- expression：绑定值的字符串形式。例如 v-my-directive="1+1"，expression 的值是"1 + 1"。
- arg：传给指令的参数。例如 v-my-directive:foo，arg 的值是"foo"。
- modifiers：一个包含修饰符的对象。例如：v-my-directive.foo.bar，修饰符对象—modifiers 的值是{foo: true, bar: true}。

- vnode：Vue 编译生成的虚拟节点。
- oldVnode：上一个虚拟节点，仅在 update 和 componentUpdated 钩子中可用。

虽然是函数的参数，但是除了 el 之外，其他参数都应该是只读的。指令注册的时候这些参数已经存在，我们还是尽量不要修改它们。如果需要在钩子函数之间共享数据，建议通过元素的 dataset 来进行。

3.3.14.2 函数简写

大多数情况下，我们可能想在 bind 和 update 钩子上做重复动作，并且不想关心其他的钩子函数。可以这样写：

```
Vue.directive('color-swatch', function (el, binding) {
    el.style.backgroundColor = binding.value
})
```

3.3.14.3 对象字面量

如果指令需要多个值，可以传入一个 JavaScript 对象字面量。记住，指令函数能够接受所有合法类型的 JavaScript 表达式。

```
<div id="app">
    <div v-demo="{ color: 'white', text: 'hello!' }"></div>
</div>

<script type="text/javascript">
    new Vue({
        el: '#app',
        data: {
            message: 'hello!'
        },
        directive: {
```

```
            'demo': function (el, binding) {
                console.log(binding.value.color) // => "white"
                console.log(binding.value.text)  // => "hello!"
            }
        }
    })
</script>
```

3.4　过滤器

Vue.js 允许你自定义过滤器，可被用作一些常见的文本格式化。过滤器可以用在两个地方：插值和 v-bind 表达式。过滤器应该被添加在 JavaScript 表达式的尾部，由“|”分隔：

```
<!-- 差值 -->
{{ text | formatText }}
<!-- in v-bind -->
<div v-bind:id="text | formatText"></div>
```

过滤器函数总接收表达式的值（之前的操作链的结果）作为第一个参数。在这个例子中，formatText 过滤器函数将会收到 text 的值作为第一个参数。

```
new Vue({
    // ...
    filters: {
        formatText: function (value) {
            value = value ? value.toString() : ''
            return value
        }
    }
})
```

过滤器可以串联：

```
{{ text | formatA | formatB }}
```

在这个例子中，formatA 的返回值会作为第一个参数传递到 formatB 中。过滤器是 JavaScript 的函数，因此可以接收参数：

```
{{ text | formatA('arg1', arg2) }}
```

这里，formatA 被定义为接收三个参数的过滤器函数。其中 message 的值作

为第一个参数，普通字符串“arg1”作为第二个参数，表达式“arg2”取值后的值作为第三个参数。

3.5 计算属性

模板内支持表达式是非常便利的，但是它们实际上只能用于简单的运算。如果在模板中放入太多的逻辑则会让模板过重，不便于理解以及日后的维护。例如：

```
<div id="example">
  {{ message.split('').reverse().join('') }}
</div>
```

在这种情况下，模板变得不再简单和清晰，模板和 js 分离得不够彻底，显得有杂质的样子。当你需要做一些改动的时候，你需要反复仔细地去理解并确认，问题会变得更糟糕，更加费时费力。

所以对于这类的复杂逻辑，计算属性是一个很好的选择。

3.5.1 基础例子

```
<div id="example">
  <p>Original message: "{{ message }}"</p>
  <p>Computed reversed message: "{{ reversedMessage }}"</p>
</div>
var vm = new Vue({
  el: '#example',
  data: {
    message: 'Hello'
  },
  computed: {
    // a computed getter
    reversedMessage: function () {
      // `this` points to the vm instance
      return this.message.split('').reverse().join('')
    }
  }
})
```

结果：

Original message: "Hello"

Computed reversed message: "olleH"

这里我们声明了一个计算属性 reversedMessage。我们提供的函数将用作属性 vm.reversedMessage 的 getter。

```
console.log(vm.reversedMessage) // -> 'olleH'
vm.message = 'Goodbye'
console.log(vm.reversedMessage) // -> 'eybdooG'
```

计算属性是用来声明式的描述一个值依赖了其他的值。我们可以像绑定普通属性一样在模板中绑定计算属性。当你在模板里把数据绑定到一个计算属性上时，Vue 会在其依赖的任何值导致该计算属性改变时更新 DOM。这个功能非常强大，它可以让你的代码更加声明式、数据驱动并且易于维护。

3.5.2　计算属性 vs Methods

你可能会有一个疑问，我们似乎也可以通过调用表达式中的 method 来达到同样的效果，那么计算属性和 method 又有什么区别呢？下面来看一个示例：

```
<p>Reversed message: "{{ reversedMessage() }}"</p>
// in component
methods: {
  reversedMessage: function () {
    return this.message.split('').reverse().join('')
  }
}
```

由上述的示例我们可以看到将同一函数定义为一个 method 而不是一个计算属性，也可以达到与计算属性相同的结果。然而，不同的是计算属性是基于它们的依赖进行缓存的。计算属性只有在它的相关依赖发生改变时才会重新求值。这就意味着只要 message 还没有发生改变，多次访问 reversedMessage 计算属性会立即返回之前的计算结果，而不必再次执行函数。这时我们就要说到**计算属性缓存**的问题。

我们为什么需要缓存？假设我们有一个性能开销比较大的计算属性 A，它需要遍历一个极大的数组和做大量的计算。然后我们可能有其他的计算属性依赖于 A。如果没有缓存，我们将不可避免地多次执行 A 的 getter！

3.5.3　计算属性缓存

在之前，计算属性仅仅体现为一个取值的行为——每次你访问它的时候，getter 都会重新求值。后来，对此做了改进——计算属性的值会被缓存，只有在其

某个反应依赖改变时才会重新计算。

这也同样意味着下面的计算属性将不再更新，因为 Date.now()不是响应式依赖，因为它没有和 Vue 的数据观察系统发生任何关系。因此，当你在程序中访问 vm.now 时，你会发现除非 vm.msg 触发了一次重新计算，否则时间戳始终是相同的值。

```
computed: {
  now: function () {
    return Date.now()
  }
}
```

相比而言，只要发生重新渲染，method 调用总会执行该函数。

而有的时候你需要保留简单获取数据的模式，每次你访问 vm.example 的时候都希望触发重新计算。你可以为一个特殊的计算属性开关缓存支持：

```
computed: {
  now: function () {
    cache: false,
    return Date.now()
  }
}
```

现在，每次你访问 vm.now 的时候，时间戳都会及时更新。然而，要注意这只发生在 JavaScript 程序内部访问的时候，数据绑定还是依赖驱动的。当你在模板中绑定一个{{now}}的计算属性时，DOM 只会在反应式依赖改变时才会更新。如果不希望有缓存，则可以用 method 替代。

3.5.4 Computed 属性 vs Watched 属性

Vue 了一种更通用的方式来观察和响应 Vue 实例上的数据变动：watch 属性。当你有一些数据需要随着其他数据变动而变动时，你很容易滥用 watch——特别是如果你之前使用过 AngularJS。然而，通常更好的想法是使用 computed 属性而不是命令式的 watch 回调。细想一下以下这个例子：

```
<div id="demo">{{ fullName }}</div>
var vm = new Vue({
  el: '#demo',
  data: {
    firstName: 'Foo',
```

```
    lastName: 'Bar',
    fullName: 'Foo Bar'
  },
  watch: {
    firstName: function (val) {
      this.fullName = val + ' ' + this.lastName
    },
    lastName: function (val) {
      this.fullName = this.firstName + ' ' + val
    }
  }
})
```

上面代码是命令式的和重复的。将它与 computed 属性的版本进行比较：

```
var vm = new Vue({
  el: '#demo',
  data: {
    firstName: 'Foo',
    lastName: 'Bar'
  },
  computed: {
    fullName: function () {
      return this.firstName + ' ' + this.lastName
    }
  }
})
```

变得好多了，不是吗？

3.5.5　计算 setter

计算属性默认只有 getter，不过在需要时你也可以提供一个 setter：

```
// ...
computed: {
  fullName: {
    // getter
    get: function () {
      return this.firstName + ' ' + this.lastName
    },
    // setter
    set: function (newValue) {
      var names = newValue.split(' ')
      this.firstName = names[0]
```

```
        this.lastName = names[names.length - 1]
      }
    }
  }
  // ...
```

现在再运行 vm.fullName = 'John Doe'时，setter 会被调用，vm.firstName 和 vm.lastName 也相应地会被更新。

3.6 观察者 Watchers

虽然计算属性在大多数情况下更合适，但有时也需要一个自定义的 watcher 来维持。这就说明了为什么 Vue 提供了一个更加通用的方法，通过 watch 选项来响应数据的变化。当你想要在数据变化响应时，执行异步操作或开销较大的操作，这是很有用的。

例如：

```
  <div id="watch-example">
    <p>
      Ask a yes/no question:
      <input v-model="question">
    </p>
    <p>{{ answer }}</p>
  </div>
  <!-- Since there is already a rich ecosystem of ajax libraries    -->
  <!-- and collections of general-purpose utility methods, Vue core -->
  <!-- is able to remain small by not reinventing them. This also   -->
  <!-- gives you the freedom to just use what you're familiar with. -->
  <script
src="https://unpkg.com/axios@0.12.0/dist/axios.min.js"></script>
  <script
src="https://unpkg.com/lodash@4.13.1/lodash.min.js"></script>
  <script>
  var watchExampleVM = new Vue({
    el: '#watch-example',
    data: {
      question: '',
      answer: 'I cannot give you an answer until you ask a question!'
    },
    watch: {
      // 如果 question 发生改变，这个函数就会运行
      question: function (newQuestion) {
```

```
      this.answer = 'Waiting for you to stop typing...'
      this.getAnswer()
    }
  },
  methods: {
    // _.debounce 是一个通过 lodash 限制操作频率的函数。
    // 在这个例子中，我们希望限制访问 yesno.wtf/api 的频率
    // ajax 请求直到用户输入完毕才会发出
    // 学习更多关于 _.debounce function (and its cousin
    // _.throttle)，参考: https://lodash.com/docs#debounce
    getAnswer: _.debounce(
      function () {
        if (this.question.indexOf('?') === -1) {
          this.answer = 'Questions usually contain a question mark. ;-)'
          return
        }
        this.answer = 'Thinking...'
        var vm = this
        axios.get('https://yesno.wtf/api')
          .then(function (response) {
            vm.answer = _.capitalize(response.data.answer)
          })
          .catch(function (error) {
            vm.answer = 'Error! Could not reach the API. ' + error
          })
      },
      // 这是我们为用户停止输入等待的毫秒数
      500
    )
  }
})
</script>
```

结果如下：

Ask a yes/no question:

I cannot give you an answer until you ask a question!

在这个示例中，使用 watch 选项允许我们执行异步操作（访问一个 API），限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这是计算属性无法做到的。

除了 watch 选项之外，还可以使用 vm.$watch API 命令。

3.7 组件的功能与使用

组件（Component）是 Vue.js 最强大的功能之一，组件化编程允许我们使用小型、独立和通常可复用的组件构建大型应用。几乎任意类型的应用界面都可以抽象为一个组件树，如图 3-4 所示。

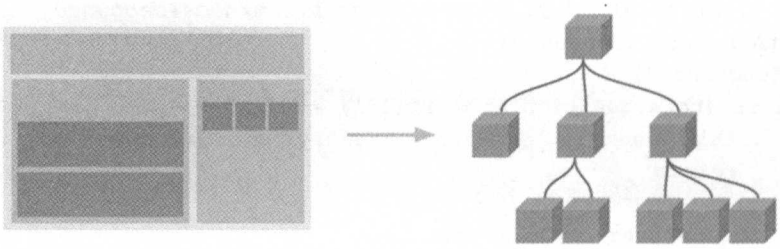


图 3-4

组件可以扩展 HTML 元素，封装可重用的代码。在较高层面上，组件是自定义元素，Vue.js 的编译器为它添加特殊功能。在有些情况下，组件也可以是原生 HTML 元素的形式。

3.7.1 使用组件

3.7.1.1 注册

在 Vue 里，一个组件在本质上是拥有预定义选项的 Vue 实例，在 Vue 中注册组件很简单，例如：

```
Vue.component('my-component', {
  // 选项
})
```

组件在注册之后，便可以在父实例的模块中以自定义元素<my-component></my-component>的形式使用。要确保在初始化根实例之前注册了组件：

```
<div id="app">
    <my-component></my-component>
</div>
```

```
<script type="text/javascript">
    Vue.component('my-component', {
        template: '<div>test component!</div>'
    })
    new Vue({
        el: '#app'
    })
</script>
```

渲染为：

```
<div id="example">
    <div>test component!</div>
</div>
```

当然，组件和指令一样，也可以局部注册。通过使用组件实例选项注册，可以使组件仅在另一个实例/组件的作用域中可用：

```
<div id="app">
    <my-component></my-component>
</div>
<script type="text/javascript">
    new Vue({
        el: '#app',
        components: {
            'my-component': {
                template: '<div>test component!</div>'
            }
        }
    })
</script>
```

3.7.1.2　DOM 模板解析说明

当使用 DOM 作为模板时你会受到 HTML 的一些限制，因为 Vue 只有在浏览器解析和标准化 HTML 后才能获取模板内容。尤其像这些元素<ul>，<ol>，<table>，<select>限制了能被它包裹的元素，而一些像<option>这样的元素只能出现在某些其他元素内部，例如：

```
<ul>
    <my-component>...</my-component>
</ul>
```

组件<my-component>被认为是无效的内容，因此在渲染时会导致错误。变通的方案是使用特殊的 is 属性：

```
<table>
    <tr is="my-row"></tr>
</table>
```

当然使用来自以下来源的字符串模板则不会有限制：

- <script type="text/x-template">
- JavaScript 内联模板字符串
- .Vue 组件

因此，有必要的话，请使用字符串模板。

3.7.1.3 data 必须是函数

通过 Vue 构造器传入的各种选项大多数都可以在组件里用。data 是一个例外，它必须是函数。例如：

```
Vue.component('my-component', {
    template: '<span>{{ message }}</span>',
    data: {
        message: 'hello'
    }
})
```

那么，Vue 报错，正确的方案如下：

```
Vue.component('my-component', {
    template: '<span>{{ message }}</span>',
    data: function() {
        return {
            message: 'hello'
        }
    }
})
```

3.7.2 组件开发

正如本章开头所说，页面是一个组件树结构，那势必会有父子关系。父子组件之间需要相互通信：父组件要给子组件传递数据，子组件需要将它内部发生的事情告知父组件。

在 Vue 中，父组件通过 props 向下传递数据给子组件，子组件通过 events 给父组件发送消息，如图 3-5 所示。

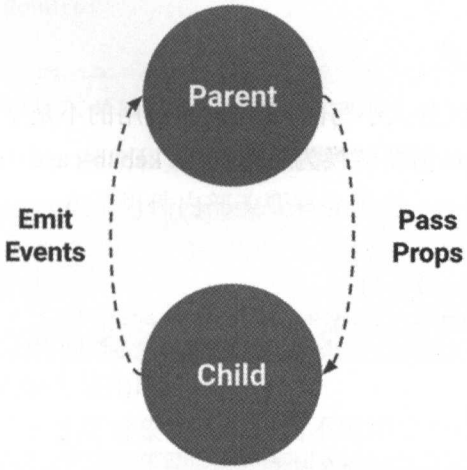


图 3-5

3.7.2.1　Props

组件实例的作用域是孤立的。要让子组件使用父组件的数据，我们需要通过子组件的 props 选项，声明期望获得的数据。

```
Vue.component('my-component', {
    // 声明 props
    props: ['message'],
    // 像 data 一样，prop 可以用在模板内
    template: '<span>{{ message }}</span>'
})
```

然后我们可以这样向它传入一个普通字符串：

```
<child message="hello!"></child>
```

渲染的结果为：

```
<span>hello</span>
```

如果想动态绑定父组件的数据到子模板的 props，用讲过的 v-bind 指令即可以实现：

```
<div>
    <my-component v-bind:message="myMessage"></my-component>
</div>
```

3.7.2.2 命名规范

HTML 特性是不区分大小写的。所以，当使用的不是字符串模板，camelCased（驼峰式）命名的 props 需要转换为相对应的 kebab-case（短横线隔开式）命名：

```
Vue.component('child', {
    // JS 里可以使用 camelCased 命名方式
    props: ['myMessage'],
    template: '<span>{{ myMessage }}</span>'
})
```

模板里需这样写：

```
// 需要转换为 kebab-case 命名方式
<child my-message="hello!"></child>
```

如果你使用字符串模板，则没有这些限制。

3.7.2.3 字面量语法与动态语法

初学者常犯的一个错误是使用字面量语法传递数值：

```
<!-- 传递了一个字符串 "1" -->
<my-component num="1"></my-component>
```

因为它是一个字面 props，它的值是字符串"1"而不是 number。如果想传递一个实际的 number，需要使用 v-bind，从而让它的值被当作 JavaScript 表达式计算：

```
<!-- 传递实际的 number -->
<my-component v-bind:num="1"></my-component>
```

3.7.2.4 单向数据流

props 是单向绑定的：当父组件的属性变化时，将传导给子组件，但是不会反过来。这是为了防止子组件无意修改了父组件的状态。

另外，每次父组件更新时，子组件的所有 props 都会更新为最新值。所以也不应该在子组件内部改变 props。

如果想用 props 传入的数据，我们需要在组件内部定义一个局部变量，用 props 的值初始化它：

```
props: ['initialCounter'],
data: function () {
  return { counter: this.initialCounter }
}
```

因为在 JavaScript 中，对象和数组是引用类型，指向同一个内存空间，如果 props 是一个对象或数组，在子组件内部改变它会影响父组件的状态。

3.7.2.5　Props 验证

我们可以为组件的 props 指定验证规格。如果传入的数据不符合规格，Vue 会发出警告。当组件给其他人使用时，这很有用。

要指定验证规格，需要用对象的形式，而不能用字符串数组：

```
Vue.component('example', {
  props: {
    // 基础类型检测 (`null` 意思是任何类型都可以)
    propA: Number,
    // 多种类型
    propB: [String, Number],
    // 必传且是字符串
    propC: {
      type: String,
      required: true
    },
    // 数字，有默认值
    propD: {
      type: Number,
      default: 100
    },
    // 数组/对象的默认值应当由一个工厂函数返回
    propE: {
      type: Object,
      default: function () {
        return { message: 'hello' }
      }
    },
    // 自定义验证函数
    propF: {
      validator: function (value) {
```

```
                return value > 10
            }
        }
    }
})
```

type 可以是下面原生构造器：

- String
- Number
- Boolean
- Function
- Object
- Array
- Symbol

type 也可以是一个自定义构造器函数，使用 instanceof 检测。

当 prop 验证失败，Vue 会抛出警告（如果使用的是开发版本）。注意 props 会在组件实例创建之前进行校验。

3.7.3 非 Props 属性

所谓非 props 属性，就是它可以直接传入组件，而不需要定义相应的 props。因为组件的开发者并不总能预见到组件被使用的场景，所以，组件可以接收任意传入的属性，这些属性都会被添加到组件的根元素上。

3.7.4 自定义事件

我们之前讲过，props 是单项数据流的传输，提供父组件向子组件的通信，如果想实现子组件向父组件通信，需要用到自定义事件系统。

3.7.4.1 使用 v-on 绑定自定义事件

每个 Vue 实例都实现了事件接口，即：

- 使用$on(eventName)监听事件；
- 使用$emit(eventName)触发事件。

另外，父组件可以在使用子组件的地方直接用 v-on 来监听子组件触发的事件。注意，不能用$on 侦听子组件释放的事件，而必须在模板里直接用 v-on 绑定。

下面是一个例子：

```
<div id="counter-event-example" class="demo">
    <p>{{ total }}</p>
    <button-counter v-on:increment="incrementTotal"></button-counter>
    <button-counter v-on:increment="incrementTotal"></button-counter>
</div>
<script type="text/javascript">
    Vue.component('button-counter', {
        template: '<button v-on:click= "incrementCounter"> {{ counter }}
</button>',
        data: function () {
            return {
                counter: 0
            }
        },
        methods: {
            incrementCounter: function () {
                this.counter += 1
                this.$emit('increment')
            }
        },
    })
    new Vue({
        el: '#counter-event-example',
        data: {
            total: 0
        },
        methods: {
            incrementTotal: function () {
                this.total += 1
            }
        }
    })
</script>
```

在本例中，子组件已经和它外部完全解耦了。它所做的只是报告自己的内部事件，至于父组件是否使用则与它无关。

3.7.4.2　给组件绑定原生事件

有时候，你可能想在某个组件的根元素上监听一个原生事件。可以使用.native 修饰 v-on。例如：

```
<my-component v-on:click.native="doTheThing"></my-component>
```

3.7.4.3 .sync 修饰符

在一些情况下，我们可能会需要对一个 prop 进行“双向绑定”。我们可以使用.sync 修饰符，但是它只是一个编译时的语法糖。它会被扩展为一个自动更新父组件属性的 v-on 侦听器。例如：

```
<comp :foo.sync="bar"></comp>
```

会被扩展为：

```
<comp :foo="bar" @update:foo="val => bar = val"></comp>
```

当子组件需要更新 foo 的值时，它需要显式地触发一个更新事件：

```
this.$emit('update:foo', newValue)
```

3.7.4.4 使用自定义事件的表单输入组件

自定义事件可以用来创建自定义的表单输入组件，使用 v-model 来进行数据双向绑定：

```
<input v-model="something">
```

其实是下一代码的语法糖：

```
<input
   v-bind:value="something"
   v-on:input="something = $event.target.value">
```

在组件中使用时，它相当于下面的简写：

```
<custom-input
   v-bind:value="something"
   v-on:input="something = arguments[0]">
</custom-input>
```

要让组件的 v-model 生效，它应该是(2.2.0+)：

- 接受一个 value 属性；
- 在有新的值时触发 input 事件。

我们来看一个非常简单的货币输入的自定义控件：

```
<div id="currency-input-example" class="demo">
```

```
    <currency-input v-model="price"></currency-input>
</div>
<script>
    Vue.component('currency-input', {
        template: '\
            <span>\
                $\
                <input\
                    ref="input"\
                    v-bind:value="value"\
                    v-on:input="updateValue($event.target.value)">\
            </span>',
        props: ['value'],
        methods: {
            // 不是直接更新值，而是使用此方法来对输入值进行格式化和位数限制
            updateValue: function (value) {
                var formattedValue = value.trim()
                // 如果值不统一，手动覆盖以保持一致
                if (formattedValue !== value) {
                    this.$refs.input.value = formattedValue
                }
                // 通过 input 事件发出数值
                this.$emit('input', Number(formattedValue))
            }
        }
    })
    new Vue({
        el: '#currency-input-example',
        data: { price: '' }
    })
</script>
```

3.7.4.5　非父子组件通信

有时候两个组件也需要通信（非父子关系）。在简单的场景下，可以使用一个空的 Vue 实例作为中央事件总线：

```
var bus = new Vue()
// 触发组件 A 中的事件
bus.$emit('id-selected', 1)
// 在组件 B 创建的钩子中监听事件
bus.$on('id-selected', function (id) {
    // ...
```

```
})
```

在复杂的情况下，我们应该考虑专门的状态管理模式，例如我们之后会讲到的 Vuex。

3.7.5 Slots 内容分发

在使用组件时，我们常常要像以下这样组合它们：

```
<app>
    <app-header></app-header>
    <app-footer></app-footer>
</app>
```

在这种情况下，<App>组件不知道它会收到什么内容。这是由使用<App>的父组件决定的，同时<App>组件有它自己的模板。

为了让组件可以组合，我们需要一种方式来混合父组件的内容与子组件自己的模板。这个过程被称为内容分发。

3.7.5.1 单个 Slot

除非子组件模板包含至少一个<slot>插口，否则父组件的内容将会被丢弃。当子组件模板只有一个没有属性的 slot 时，父组件整个内容片段将插入到 slot 所在的 DOM 位置，并替换掉 slot 标签本身。

最初在<slot>标签中的任何内容都被视为备用内容。备用内容在子组件的作用域内编译，并且只有在宿主元素为空，且没有要插入的内容时才显示备用内容。

假定 my-component 组件有下面模板：

```
<div>
    <h2>我是子组件的标题</h2>
    <slot>
        只有在没有要分发的内容时才会显示。
    </slot>
</div>
```

父组件模板：

```
<div>
    <h1>我是父组件的标题</h1>
    <my-component>
        <p>这是一些初始内容</p>
```

```
        <p>这是更多的初始内容</p>
    </my-component>
</div>
```

渲染结果：

```
<div>
    <h1>我是父组件的标题</h1>
    <div>
        <h2>我是子组件的标题</h2>
        <p>这是一些初始内容</p>
        <p>这是更多的初始内容</p>
    </div>
</div>
```

3.7.5.2　具名 Slot

<slot>元素可以用一个特殊的属性 name 来配置如何分发内容。多个 slot 可以有不同的名字。具名 slot 将匹配内容片段中有对应 slot 特性的元素。

仍然可以有一个匿名 slot，它是**默认 slot**，作为找不到匹配的内容片段的备用插槽。如果没有默认的 slot，这些找不到匹配的内容片段将被抛弃。

例如，假定我们有一个 App-layout 组件，它的模板为：

```
<div class="container">
    <header>
        <slot name="header"></slot>
    </header>
    <main>
        <slot></slot>
    </main>
    <footer>
        <slot name="footer"></slot>
    </footer>
</div>
```

父组件模板：

```
<App-layout>
    <h1 slot="header">这里可能是一个页面标题</h1>
    <p>主要内容的一个段落。</p>
    <p>另一个主要段落。</p>
    <p slot="footer">这里有一些联系信息</p>
</App-layout>
```

渲染结果为：

```
<div class="container">
    <header>
        <h1>这里可能是一个页面标题</h1>
    </header>
    <main>
        <p>主要内容的一个段落。</p>
        <p>另一个主要段落。</p>
    </main>
    <footer>
        <p>这里有一些联系信息</p>
    </footer>
</div>
```

在组合组件时，内容分发 API 是非常有用的机制。

3.7.5.3 作用域插槽（2.1.0+）

作用域插槽是一种特殊类型的插槽，用做使用一个（能够传递数据到）可重用模板替换已渲染元素。

在子组件中，只需将数据传递到插槽，就像你将 props 传递给组件一样：

```
<div class="child">
    <slot text="hello from child"></slot>
</div>
```

在父级组件中，具有特殊属性 scope 的<template>元素必须存在，表示它是作用域插槽的模板。scope 的值对应一个临时变量名，此变量接收从子组件中传递的 props 对象：

```
<div class="parent">
    <child>
        <template scope="props">
            <span>hello from parent</span>
            <span>{{ props.text }}</span>
        </template>
    </child>
</div>
```

如果我们渲染以上结果，得到的输出会是：

```
<div class="parent">
```

```
    <div class="child">
        <span>hello from parent</span>
        <span>hello from child</span>
    </div>
</div>
```

作用域插槽更具代表性的用例是列表组件，允许组件自定义应该如何渲染列表每一项：

```
<my-awesome-list :items="items">
    <!-- 作用域插槽也可以是具名的 -->
    <template slot="item" scope="props">
        <li class="my-fancy-item">{{ props.text }}</li>
    </template>
</my-awesome-list>
```

列表组件的模板：

```
<ul>
    <slot name="item"
        v-for="item in items"
        :text="item.text">
        <!-- 这里写入备用内容 -->
    </slot>
</ul>
```

3.7.6　动态组件

通过使用保留的<component>元素，动态地绑定到它的 is 特性，我们让多个组件可以使用同一个挂载点，并动态切换：

```
<component v-bind:is="currentView">
  <!-- 组件在 vm.currentview 变化时改变! -->
</component>

<script type="text/javascript">
    var vm = new Vue({
        el: '#example',
        data: {
            currentView: 'home'
        },
        components: {
            home: { /* ... */ },
            posts: { /* ... */ },
```

```
            archive: { /* ... */ }
        }
    })
</script>
```

也可以直接绑定到组件对象上：

```
var Home = {
    template: '<p>Welcome home!</p>'
}

var vm = new Vue({
    el: '#example',
    data: {
        currentView: Home
    }
})
```

如果把切换出去的组件保留在内存中，可以保留它的状态或避免重新渲染。为此可以添加一个 keep-alive 指令参数：

```
<keep-alive>
    <component :is="currentView">
        <!-- 非活动组件将被缓存! -->
    </component>
</keep-alive>
```

3.7.7 组件的其他知识

3.7.7.1 子组件索引

尽管有 props 和 events，但是有时仍然需要在 JavaScript 中直接访问子组件。为此可以使用 ref 为子组件指定一个索引 ID。例如：

```
<div id="parent">
  <user-profile ref="profile"></user-profile>
</div>

<script type="text/javascript">
    var parent = new Vue({
        el: '#parent'
    })
    // 访问子组件
```

```
    var child = parent.$refs.profile
</script>
```

当 ref 和 v-for 一起使用时，ref 是一个数组，包含相应的子组件。$refs 只在组件渲染完成后才填充，并且它是非响应式的。它仅仅作为一个直接访问子组件的应急方案——应当避免在模板或计算属性中使用$refs。

3.7.7.2　异步组件

在大型应用中，我们可能需要将应用拆分为多个小模块，按需从服务器下载。为了让事情更简单，Vue.js 允许将组件定义为一个工厂函数，动态地解析组件的定义。Vue.js 只在组件需要渲染时触发工厂函数，并且把结果缓存起来，用于后面的再次渲染。例如：

```
Vue.component('async-example', function (resolve, reject) {
    setTimeout(function () {
        resolve({
            template: '<div>I am async!</div>'
        })
    }, 1000)
})
```

工厂函数接收一个 resolve 回调，在收到从服务器下载的组件定义时调用。也可以调用 reject(reason)指示加载失败。这里 setTimeout 只是为了演示。怎么获取组件完全由你决定。推荐配合使用 Webpack 的代码分割功能。

3.7.7.3　高级异步组件

异步组件的工厂函数也可以返回一个如下的对象：

```
const AsyncComp = () => ({
    // 需要加载的组件. 应当是一个 Promise
    component: import('./MyComp.Vue'),
    // loading 时应当渲染的组件
    loading: LoadingComp,
    // 出错时渲染的组件
    error: ErrorComp,
    // 渲染 loading 组件前的等待时间。默认：200ms.
    delay: 200,
    // 最长等待时间。超出此时间则渲染 error 组件。默认：Infinity
    timeout: 3000
})
```

注意，当一个异步组件被作为 Vue-router 的路由组件使用时，这些高级选项都是无效的，因为在路由切换前就会提前加载所需要的异步组件。另外，如果你要在路由组件中使用上述写法，需要使用 Vue-router2.4.0+。

3.7.7.4 组件命名约定

当注册组件(或者 props)时，可以使用 kebab-case，camelCase，或 PascalCase。

```
// 在组件定义中
components: {
    // 使用 kebab-case 形式注册
    'kebab-cased-component': { /* ... */ },
    // register using camelCase
    'camelCasedComponent': { /* ... */ },
    // register using PascalCase
    'PascalCasedComponent': { /* ... */ }
}
```

在 HTML 模板中，请使用 kebab-case 形式：

```
<!-- 在 HTML 模板中始终使用 kebab-case -->
<kebab-cased-component></kebab-cased-component>
<camel-cased-component></camel-cased-component>
<pascal-cased-component></pascal-cased-component>
```

当使用字符串模式时，可以不受 HTML 的 case-insensitive 限制。这意味着在模板中，你可以使用下面的方式来引用你的组件：

- kebab-case；
- camelCase 或 kebab-case 如果组件已经被定义为 camelCase；
- kebab-case，camelCase 或 PascalCase，如果组件已经被定义为 PascalCase。

```
components: {
    'kebab-cased-component': { /* ... */ },
    camelCasedComponent: { /* ... */ },
    PascalCasedComponent: { /* ... */ }
}
<kebab-cased-component></kebab-cased-component>

<camel-cased-component></camel-cased-component>
<camelCasedComponent></camelCasedComponent>

<pascal-cased-component></pascal-cased-component>
```

```
<pascalCasedComponent></pascalCasedComponent>
<PascalCasedComponent></PascalCasedComponent>
```

这意味着 PascalCase 是最通用的声明约定，而 kebab-case 是最通用的使用约定。

如果组件未经 slot 元素传递内容，你甚至可以在组件名后使用 / 使其自闭合：

```
<my-component/>
```

当然，这只在字符串模板中有效。因为自闭的自定义元素是无效的 HTML，浏览器原生的解析器也无法识别它。

3.7.7.5　递归组件

组件在它的模板内可以递归地调用自己，不过，只有当它有 name 选项时才可以：

```
name: 'unique-name-of-my-component'
```

当你利用 Vue.component 全局注册了一个组件，全局的 ID 作为组件的 name 选项，被自动设置。

```
Vue.component('unique-name-of-my-component', {
  // ...
})
```

如果你不谨慎，递归组件可能导致死循环：

```
name: 'stack-overflow',
template: '<div><stack-overflow></stack-overflow></div>'
```

上面组件会导致一个错误“max stack size exceeded”，所以要确保递归调用有终止条件（比如递归调用时使用 v-if 并让他最终返回 false）。

3.7.7.6　内联模板

如果子组件有 inline-template 特性，组件将把它的内容当作它的模板，而不是把它当作分发内容。这让模板更加灵活：

```
<my-component inline-template>
  <div>
    <p>These are compiled as the component's own template.</p>
    <p>Not parent's transclusion content.</p>
```

```
    </div>
</my-component>
```

但是，inline-template 让模板的作用域难以理解。最佳实践是使用 template 选项在组件内定义模板或者在.vue 文件中使用 template 元素。

3.7.7.7 X-Templates

另一种定义模板的方式是在 JavaScript 标签里使用 text/x-template 类型，并且指定一个 id。例如：

```
<script type="text/x-template" id="hello-world-template">
    <p>Hello hello hello</p>
</script>
<script type="text/javascript">
    Vue.component('hello-world', {
        template: '#hello-world-template'
    })
</script>
```

这在有很多模板或者小的应用中有用，否则应该避免使用，因为它将模板和组件的其他定义隔离了。

3.7.7.8 对低开销的静态组件使用 v-once

尽管在 Vue 中渲染 HTML 很快，不过当组件中包含大量静态内容时，可以考虑使用 v-once 将渲染结果缓存起来，就像这样：

```
Vue.component('terms-of-service', {
    template: '\
        <div v-once>\
            <h1>Terms of Service</h1>\
            ... a lot of static content ...\
        </div>'
})
```

3.8 继承与混合

众所周知，Vue 组件自身是以对象的方式声明的，既然是对象，考虑到代码复用的需求，Vue 也提供了几种不同的组件间代码复用的能力，分别是继承（options 里的 extends、Vue.extend）和混合（options 里的 mixins）。

3.8.1　Vue.extend

Vue.extend()基础 Vue 构造器，创建一个“子类”。参数是一个包含组件选项的对象。

data 选项是特例，需要注意：在 Vue.extend()中它必须是函数，而在 new Vue() data 选项中可以是对象，这是因为 new Vue()是一个 Vue 应用的根，它返回 VM 对象而不是组件的构造函数，所以 data 选项不会被复用。而 Vue.extend()返回的是组件的构造函数，它的 data 选项会被每一个组件实例给复用，所以 data 必须是函数，这样每个实例的 data 指向的对象都是独立的。

3.8.2　options 里的 extends

允许声明扩展另一个组件（可以是一个简单的选项对象或构造函数），而无需使用 Vue.extend。这主要是为了便于扩展单文件组件。

options 里的 mixins mixins 选项接受一个混合对象的数组。这些混合实例对象可以像正常的实例对象一样包含选项，他们将在 Vue.extend()里最终选择使用相同的选项进行逻辑合并。举例：如果你混合包含一个钩子而创建组件本身也有一个，两个函数将被调用。Mixin 钩子按照传入顺序依次调用并在调用组件自身的钩子之前被调用。

3.8.3　源码分析

我们来看一下 Vue 的源码，下面是函数 mergeOptions，文件地址是 src/core/util/options.js，这个是 Vue 继承最关键的函数。

```
export function mergeOptions (
  parent: Object,
  child: Object,
  vm?: Component
): Object {
  if (process.env.NODE_ENV !== 'production') {
    checkComponents(child)
  }
  normalizeProps(child)
  normalizeDirectives(child)
  const extendsFrom = child.extends
  if (extendsFrom) { // 和下面的mixins处理方式上没有什么不同
    parent = typeof extendsFrom === 'function'
```

```
        ? mergeOptions(parent, extendsFrom.options, vm)
        : mergeOptions(parent, extendsFrom, vm)
    }
    if (child.mixins) {
      for (let i = 0, l = child.mixins.length; i < l; i++) {
        let mixin = child.mixins[i]
        if (mixin.prototype instanceof Vue) {
          mixin = mixin.options
        }
        parent = mergeOptions(parent, mixin, vm)
      }
    }
    const options = {}
    let key
    for (key in parent) {
      mergeField(key)
    }
    for (key in child) {
      if (!hasOwn(parent, key)) {
        mergeField(key)
      }
    }
    function mergeField (key) {
      const strat = strats[key] || defaultStrat
      options[key] = strat(parent[key], child[key], vm, key)
    }
    return options
  }
```

我们可以看到 option 里的 extends 和 options 在处理方式上没有什么不同，唯一的不同是 extends 的优先级要比 mixins 要高。extends 会先和 Parent 合并，然后才是 mixins[0],mixins[1]，那优先级具体指的是什么呢？我们看下合并策略就明白了。

3.8.4 合并策略

Vue 有自身的合并策略，不同的 options 选项有不同的合并策略，一共有 6 种，分别是：

- lifeCycle
- other hash object (props, methods, computed)
- data

- assets
- default
- el、propsData

3.8.4.1　lifeCycle

首先是 lifeCycle，lifeCycle 的合并策略是

```
function mergeHook (
  parentVal: ?Array<Function>,
  childVal: ?Function | ?Array<Function>
): ?Array<Function> {
  return childVal
    ? parentVal
      ? parentVal.concat(childVal)
      : Array.isArray(childVal)
        ? childVal
        : [childVal]
    : parentVal
}

config._lifecycleHooks.forEach(hook => {
  strats[hook] = mergeHook
})
```

lifeCycle 的合并策略比较简单，就是将同一个 lifeCycle 合并成一个数组，父组件的 lifeCycle 的顺序在子组件前面，先执行父组件的 lifeCycle 再执行子组件的 lifeCycle。

3.8.4.2　other hash object (props, methods, computed)

```
strats.props =
strats.methods =
strats.computed = function (parentVal: ?Object, childVal: ?Object):
?Object {
  if (!childVal) return parentVal
  if (!parentVal) return childVal
  const ret = Object.create(null)
  extend(ret, parentVal)
  extend(ret, childVal)
  return ret
}
```

对于这类对象的合并策略也非常简单，就是父子组件的所有 key 值合并，并且子组件的覆盖父组件的。extend 方法就是简单的赋值而已！

```
/**
 * Mix properties into target object.
 */
export function extend (to: Object, _from: ?Object): Object {
  for (const key in _from) {
    to[key] = _from[key]
  }
  return to
}
```

3.8.4.3 data

```
strats.data = function (
  parentVal: any,
  childVal: any,
  vm?: Component
): ?Function {
  if (!vm) {
    // in a Vue.extend merge, both should be functions
    if (!childVal) {
      return parentVal
    }
    if (typeof childVal !== 'function') {
      process.env.NODE_ENV !== 'production' && warn(
        'The "data" option should be a function ' +
        'that returns a per-instance value in component ' +
        'definitions.',
        vm
      )
      return parentVal
    }
    if (!parentVal) {
      return childVal
    }
    // when parentVal & childVal are both present,
    // we need to return a function that returns the
    // merged result of both functions... no need to
    // check if parentVal is a function here because
    // it has to be a function to pass previous merges.
```

```
    return function mergedDataFn () {
      return mergeData(
        childVal.call(this),
        parentVal.call(this)
      )
    }
  } else if (parentVal || childVal) {
    return function mergedInstanceDataFn () {
      // instance merge
      const instanceData = typeof childVal === 'function'
        ? childVal.call(vm)
        : childVal
      const defaultData = typeof parentVal === 'function'
        ? parentVal.call(vm)
        : undefined
      if (instanceData) {
        return mergeData(instanceData, defaultData)
      } else {
        return defaultData
      }
    }
  }
}
/**
 * Helper that recursively merges two data objects together.
 */
function mergeData (to: Object, from: ?Object): Object {
  if (!from) return to
  let key, toVal, fromVal
  const keys = Object.keys(from)
  for (let i = 0; i < keys.length; i++) {
    key = keys[i]
    toVal = to[key]
    fromVal = from[key]
    if (!hasOwn(to, key)) {
      set(to, key, fromVal)
    } else if (isPlainObject(toVal) && isPlainObject(fromVal)) {
      mergeData(toVal, fromVal)
    }
  }
  return to
}
```

data 的合并策略稍微复杂点，因为 data 在根组件可以直接传入一个对象，而

在子组件只能以函数的形式存在，所以一上来是要判断父组件和子组件的 data 选项是否是函数，如果是函数要先执行后得到返回的对象后再调用 mergeData 方法去 merge，而在 mergeData 这个方法里也就是将父组件的 data 赋值到组件上去，如果两个都是对象的话就循环调用 mergeData。

3.8.4.4 assets

```
function mergeAssets (parentVal: ?Object, childVal: ?Object): Object
{
  const res = Object.create(parentVal || null)
  return childVal
    ? extend(res, childVal)
    : res
}
```

assets 的合并策略使用了原型链，不是简单的子组件覆盖父组件，而是使用 Object.create 以父组件的值为原型建立一个原型对象，然后将子组件的值覆盖整个空的原型对象。这样有一条原型链，也就是只有在父组件的原型链上能找到 assets，在子组件上都可以直接引用。

3.8.4.5 default

```
const defaultStrat = function (parentVal: any, childVal: any): any {
  return childVal === undefined
    ? parentVal
    : childVal
}
```

一些默认的选项合并就是子组件，有就使用子组件的，不然就使用父组件。

3.8.4.6 el、propsData

```
strats.el = strats.propsData = function (parent, child, vm, key) {
  if (!vm) {
    warn(
      `option "${key}" can only be used during instance ` +
      'creation with the `new` keyword.'
    )
  }
  return defaultStrat(parent, child)
}
```

这两个和默认合并选项一样只是加了一些提示。

3.9　插件 plugin

插件通常会为 Vue 添加全局功能，当我们引用插件以后便可以按照固定的规则去使用插件的功能，在 Vue 的范畴里插件的范围没有限制，一般会有如下几种形式。

- 添加全局方法或者属性；
- 添加全局资源：指令、过滤器、过渡等；
- 通过全局 mixin 方法添加一些组件选项；
- 添加 Vue 实例方法，通过把它们添加到 Vue.prototype 上实现；
- 一个库，提供自己的 API，同时提供上面提到的一个或多个功能 Vue.js 的插件有一个公共方法 install，当我们引用插件的时候执行这个 install 方法。改方法有两个参数，第一个是 Vue 构造器，第二个参数是一个可选的选项对象。

在本例子中有一个关于点赞的插件，下面通过这个点赞插件对插件 API 进行讲解。首先在目录结构里增加一个 plugin 文件夹，文件夹里新建一个 zan.js 文件，该文件里则是点赞逻辑处理。因为是一个插件，根据 Vue.js 提供的方案，我们需要一个暴露在外面的 install 方法，因此初始结构如下：

```
export default{
   install(Vue, options){
    // do something
   }
}
```

在 install 方法里我们可以先定义一个实例方法，实例方法执行完成点赞，具体方法如下：

```
export default{
   install(Vue, options){
    // 添加实例方法
    Vue.prototype.$zan = function(ele){
     ele.style.animation = 'zan 1s ease-in';
     setTimeout(function(){
         ele.parentNode.querySelector('path').style.fill = '#ff6600';
     }, 1000);
```

```
        }
      }
  }
```

在我们事先定义好了 css 以后，通过在 Vue 原型扩展方法，在方法里传入触发元素从而完成对点赞元素的操作控制，实现点赞功能。此时，我们的点赞插件功能已经实现，接下来将这个插件引入到我们的项目里，在我们的目录结构里引入这个文件，同时使用插件即可。

```
...
// 引入插件文件
import Zan from './plugin/zan'

// 使用插件
Vue.use(Zan)
...
```

Vue.use 方法完成 install 方法的执行，此时全局 Vue 实例有了点赞功能，我们还需要在使用的地方进行传递使用即可。

```
...
<span class="zan" @click='approve($event.target)'>
...

methods: {
  approve(t){
    // 调用实例方法
     this.$zan(t);
  }
}
```

刷新页面，点击点赞按钮，实现功能。由于插件没有范围限制，我们还可以通过自定义指令的形式生成点赞功能，具体在插件里的实现方式如下：

```
export default{
   install(Vue, options){
    // 添加全局资源
    Vue.directive('zan', {
     inserted(el){
          el.onclick = function(){
               el.style.animation = 'zan 1s ease-in';
               setTimeout(function(){
```

```
                    el.parentNode.querySelector('path').style.fill    =
'#ff6600';
                }, 1000);
            }
        }
      })
    }
}
```

在使用上我们在需要的元素上绑定自定义指令即可。还有全局方法和属性定义以及 mixin 的插件开发方式，围绕点赞功能进行如下示例：

```
export default{
    install(Vue, options){
     //全局方法和属性，全局赞
     Vue.zans = function(tag, options){
      Vue.component(tag, {
          render: function(createElement){
              return createElement('h1','点赞');
          }
      });
     }

     // 注入组件 运用场景
     Vue.mixin({
      created: function(){
          var ele = this.$options.element
          if(ele){
              console.log("该元素有点赞功能~")
          }
      }
     })
    }
}
```

全局方法定义上，自定义了一个标签组件，这样一个插件可以全局使用这个自定义标签。如：

```
// 引用文件中，在引用插件以后定义一个标签名
Vue.zans('zanTip')
// 使用文件中调用
<zan-tip></zan-tip>
```

此刻，当页面中鼠标滑过点赞元素会有点赞提示文案。最后关于 mixin 的注入功能只是简单的输出 log，这个可以根据实际业务情况进行扩展开发。Vue.use 方法在使用插件的过程中，可以传入选项对象：

```
Vue.use(myPlugin, {someOption: true})
```

Vue.use 同时会阻止同一个插件的多次注册，只会对同一个插件注册一次。

读 者 圈

第 4 章　Vue 组件库

本章推荐一些成熟的第三方组件库，减少大家重复造轮子的情况。

4.1　Element

核心关键词：PC，后台页面，组件库丰富，Vue2.0。

Element 是由“饿了么”前端团队基于 Vue2.0 开发的一套 PC 端组件库，如图 4-1 所示。提供了配套的 UI，官网地址：

http://element.eleme.io/

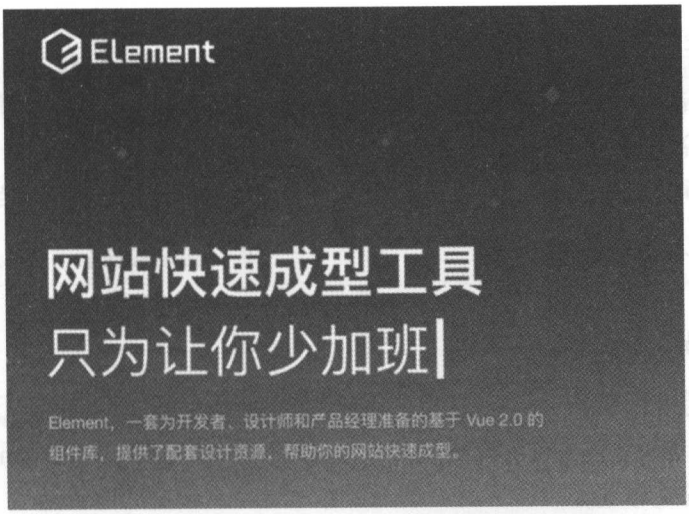


图 4-1

4.1.1　Element 的设计

Element 在设计上遵从了以下几项原则。

1. 一致性 Consistency

- 与现实生活一致：与现实生活的流程、逻辑保持一致，遵循用户习惯的语言和概念；
- 在界面中一致：所有的元素和结构需保持一致，比如：设计样式、图标和文本、元素的位置等。

2. 反馈 Feedback

- 控制反馈：通过界面样式和交互动效让用户可以清晰的感知自己的操作；
- 页面反馈：操作后，通过页面元素的变化清晰地展现当前状态。

3. 效率 Efficiency

- 简化流程：设计简洁直观的操作流程；
- 清晰明确：语言表达清晰且表意明确，让用户快速理解进而作出决策；帮助用户识别：界面简单直白，让用户快速识别而非回忆，减少用户记忆负担。

4. 可控 Controllability

- 用户决策：根据场景可给予用户操作建议或安全提示，但不能代替用户进行决策；
- 结果可控：用户可以自由的进行操作，包括撤销、回退和终止当前操作等。

4.1.2 Element 的 UI

1. 布局

主要采用了栅格化布局的方式，通过基础的 24 分栏，迅速简便地创建布局；同时，参照了 Bootstrap 的响应式设计，预设了四个响应尺寸：xs、sm、md 和 lg，可以支持响应式布局。

2. 主题

为了避免视觉传达差异，使用一套特定的调色板来规定颜色，默认主色调是蓝色，同时提供了需要在不同的场景中使用的辅助色（例如危险色表示危险的操作）和用来表现层次结构的中性色。

如果你想完全替换主题色或者部分样式，可以使用在线或者线下的主题生成工具来修改主题。

4.1.3　Element 的优缺点

Element 相较于其他 PC 组件库，最大的优势在于丰富的组件库，且每个组件扩展性很强，基本能覆盖大部分情况。

截至本书发稿日止，Element 已经更新到了 1.4.4 版本，提供了包括：表单相关组件（Radio、Checkbox、Input 等），数据展示相关组件（Table、Pagination、Tag 等），通知相关组件（Alert、Message、Notification 等），导航相关组件（NavMenu、Tabs、Steps 等），其他组件（Dialog、Card、Popover 等）等组件。

同时，饿了么团队对每个组件，都提供了清晰的使用文档和 demo，上手使用难度极低，适合作为内部平台的统一组件库。

Form 组件的相关使用文档如图 4-2 所示。

Form Attributes

参数	说明	类型	可选值	默认值
model	表单数据对象	object	–	–
rules	表单验证规则	object	–	–
inline	行内表单模式	boolean	–	false
label-position	表单域标签的位置	string	right/left/top	right
label-width	表单域标签的宽度，所有的 form-item 都会继承 form 组件的 labelWidth 的值	string	–	–
label-suffix	表单域标签的后缀	string	–	–
show-message	是否显示校验错误信息	boolean	–	true

图 4-2

Form 组件的 Demo 展示如图 4-3 所示。

不过 Element 只能做为 PC 端页面的组件库，无法快速搭建移动端应用，而且 Elment 只支持 Vue2.0 版本，如果开发者使用 Vue1.0 版本来构建项目，则无法使用 Element。

图 4-3

4.2 Mint UI

核心关键词：移动端，独立组件仓库，Vue1.0 和 Vue 2.0 都支持。

Mint UI 是由饿了么前端团队推出的基于 Vue.js 的移动端组件库。官网地址：

http://mint-ui.github.io/#!/zh-cn

Mint UI 如图 4-4 所示。

图 4-4

4.2.1 Mint UI 的特性

Mint UI 包含丰富的 CSS 和 JS 组件，能够满足日常的移动端开发需要。通过它，可以快速构建出风格统一的页面，提升开发效率。同时，Mint UI 实现了真正意义上的按需加载组件。可以只加载声明过的组件及其样式文件，无需再纠结文件体积过大。

考虑到移动端的性能门槛，Mint UI 采用 CSS3 处理各种动效，避免浏览器进行不必要的重绘和重排，从而使用户获得流畅顺滑的体验。

依托 Vue.js 高效的组件化方案，Mint UI 做到了轻量化。即使全部引入，压缩后的文件体积也仅有 30kb (JS + CSS) gzip。

4.2.2　Mint UI 的优缺点

Mint UI 拥有约 15 个 JS 组件、10 个 CSS 组件和 5 个 Form 组件，所有 JS 组件均有单独的仓库，可以独立引用，减少冗余代码的加载，优化页面性能；Mint UI 提供了通用的项目模板，新项目可以快速搭建；同时，Mint UI 对 Vue1.0 和 2.0 都有很好的支持，泛用性更强。

Mint UI 的每个组件，都拥有清晰的使用文档和 demo，上手使用难度低。

不过相比 Element，Mint UI 的组件个数偏少，且大部分组件比较简单，可能需要在项目中自己造轮子。

4.3　iView

核心关键词：PC，中后台产品，Vue1.0 和 Vue 2.0。

iView 是一套基于 Vue.js 的开源 UI 组件库，主要服务于 PC 界面的中后台产品，官网地址：

https://www.iviewui.com/docs/guide/introduce

iView 如图 4-5 所示。

图 4-5

4.3.1 iView 简介

iView 具有以下特性：

- 高质量、功能丰富；
- 友好的 API，自由灵活地使用空间；
- 事无巨细的文档；
- 细致、漂亮的 UI；
- 使用单文件的 Vue 组件化开发模式；
- 基于 npm + webpack + babel 开发，支持 ES2015。

在布局和 UI 方面，iView 和 Element 的思路相近，同样采用了 24 分栏的栅格化布局方式，UI 层面支持主题的定制，和页面元素的响应式设计。

4.3.2 iView 的优缺点

iView 的组件库丰富，Demo 和文档很清晰。最显著的特色在于 iView 提供了 iView Cli，是一个桌面客户端工具，可以用可视化的方式创建 iView 工程，适合自我动手能力不太强的开发者或团队。

iView Cli 如图 4-6 所示。

图 4-6

同时，iView 支持 Vue1.0 和 Vue2.0，泛用性更强。

4.4　Vux

核心关键词：移动端，WeUI，组件库丰富，Vue2.0。

Vux（读音 [v'ju:z]，同 views）是基于 WeUI 和 Vue(2.x)开发的移动端 UI 组件库，主要服务于微信页面。WeUI 是一套同微信原生视觉体验一致的基础样式库，由微信官方设计团队为微信内网页和微信小程序量身设计，令用户的使用感知更加统一。官网地址：

https://vux.li/#/zh-CN/README

Vux 如图 4-7 所示。

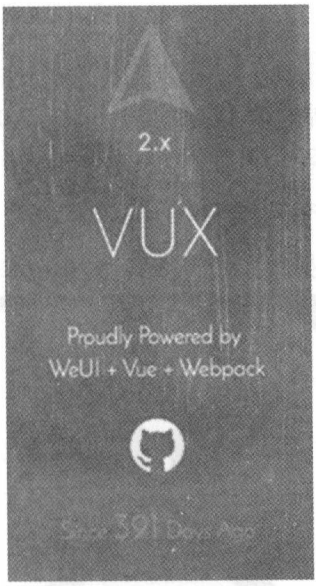


图 4-7

4.4.1　Vux 简介

移动端开发基于 webpack+Vue-loader+Vux，可以快速搭建页面，配合 Vux-loader 可以在 WeUI 的基础上定制需要的样式。

Vux-loader 保证了组件按需使用，因此不用担心最终打包了整个 Vux 的组件库代码。

Vux 并不完全依赖于 WeUI，但是尽量保持整体 UI 样式接近 WeUI 的设计规范。

4.4.2 Vux 优缺点

Vux 拥有 65+个组件，在所有组件库中都是最多的，支持使用 Vue-cli 工具和 airyland/Vux2 模板快速初始化项目。

Vux 支持用 commonJS 的方式使用微信 JS sdk，同时 UI 也是基于 WeUI 开发的，很适合开发微信内的移动端页面。

说到 Vux 的缺点，仍然是老生长谈的版本问题，由于 Vue1.0 到 Vue2.0 有了较大的升级，Vux 是无法运用到 Vue1.0 的项目中去的。

4.5 XCUI

核心关键词：PC，Vue 2.0。

XCUI 是由百度外卖前端团队推出的基于 Vue 2.0 桌面端组件库。同样，XCUI 也提供了配套的 UI 库，官网地址为：

https://wmfe.github.io/xcui/#!/home

XCUI 如图 4-8 所示。

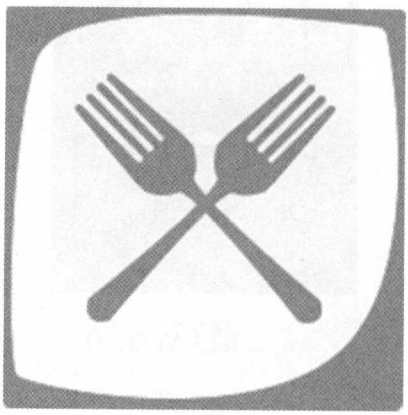


图 4-8

4.5.1 XCUI 简介

XCUI 的目标是：

- 满足桌面端页面大部分基础组件需求；
- 快速开发基于 Vue2.0 构建的项目；

- 保持小体积，无其他 js 库依赖；
- 简洁优雅。

XCUI 使用 umd 方式打包，支持各种模块加载器。

在页面根实例中引入：

```
import Vue from 'Vue'
import xcui from 'xcui'
import 'xcui/lib/xcui.css'
Vue.use(xcui);
```

或者：

```
var Vue = import 'Vue');
var xcui = import 'xcui').default;
Vue.use(xcui);
```

在页面中声明标签，即可使用。

4.5.2　XCUI 优缺点

XCUI 支持使用 FIS 作为构建工具来引入，如果业务线使用 FIS，则 XCUI 是很好的选择。

读 者 圈

第5章　官方周边库

5.1　Axios

Axios 是一个基于 promise 的 HTTP 库，可以用在浏览器和 node.js 中。

5.1.1　功能

Axios 作为一个 HTTP 库，具有以下的功能：

- 从浏览器中创建 XMLHttpRequests；
- 从 node.js 创建 http 请求；
- 支持 Promise API；
- 拦截请求和响应；
- 转换请求数据和响应数据；
- 取消请求；
- 自动转换 JSON 数据；
- 客户端支持防御 XSRF。

5.1.2　安装

使用 npm：

```
$ npm install axios
```

使用 bower：

```
$ bower install axios
```

使用 cdn：

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

5.1.3　Example

执行 GET 请求：

```
// 为给定 ID 的 user 创建请求
axios.get('/user?ID=12345')
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });

// 可选地，上面的请求可以这样做
axios.get('/user', {
    params: {
      ID: 12345
    }
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

执行 POST 请求：

```
axios.post('/user', {
    firstName: 'Fred',
    lastName: 'Flintstone'
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

执行多个并发请求：

```
function getUserAccount() {
  return axios.get('/user/12345');
}
```

```
function getUserPermissions() {
  return axios.get('/user/12345/permissions');
}

axios.all([getUserAccount(), getUserPermissions()])
  .then(axios.spread(function (acct, perms) {
    // 两个请求现在都执行完成
  }));
```

5.1.4 Axios API

可以通过向 Axios 传递相关配置来创建请求 axios(config)：

```
// 发送 POST 请求
axios({
  method: 'post',
  url: '/user/12345',
  data: {
    firstName: 'Fred',
    lastName: 'Flintstone'
  }
});
```

Axios(url[, config])：

```
// 发送 GET 请求（默认的方法）
axios('/user/12345');
```

5.1.4.1 请求方法的别名

为方便起见，为所有支持的请求方法提供了别名：

```
axios.request(config)
axios.get(url[, config])
axios.delete(url[, config])
axios.head(url[, config])
axios.post(url[, data[, config]])
axios.put(url[, data[, config]])
axios.patch(url[, data[, config]])
```

在使用别名方法时，url、method、data 这些属性都不必在配置中指定。

5.1.4.2　并发

处理并发请求的助手函数

```
axios.all(iterable)
axios.spread(callback)
```

5.1.4.3　创建实例

可以使用自定义配置新建一个 Axios 实例：

Axios.create([config])

```
var instance = axios.create({
  baseURL: 'https://some-domain.com/api/',
  timeout: 1000,
  headers: {'X-Custom-Header': 'foobar'}
});
```

5.1.4.4　实例方法

以下是可用的实例方法。指定的配置将与实例的配置合并:

```
axios#request(config)
axios#get(url[, config])
axios#delete(url[, config])
axios#head(url[, config])
axios#post(url[, data[, config]])
axios#put(url[, data[, config]])
axios#patch(url[, data[, config]])
```

5.1.5　请求配置

这些是创建请求时可以用的配置选项。只有 url 是必需的。如果没有指定 method，请求将默认使用 get 方法。

```
{
  // `url` 是用于请求的服务器 URL
  url: '/user',

  // `method` 是创建请求时使用的方法
  method: 'get', // 默认是 get
```

```
    // `baseURL` 将自动加在 `url` 前面，除非 `url` 是一个绝对 URL。
    // 它可以通过设置一个 `baseURL` 便于为 axios 实例的方法传递相对 URL
    baseURL: 'https://some-domain.com/api/',

    // `transformRequest` 允许在向服务器发送前，修改请求数据
    // 只能用在 'PUT', 'POST' 和 'PATCH' 这几个请求方法
    // 后面数组中的函数必须返回一个字符串，或 ArrayBuffer，或 Stream
    transformRequest: [function (data) {
      // 对 data 进行任意转换处理

      return data;
    }],

    // `transformResponse` 在传递给 then/catch 前，允许修改响应数据
    transformResponse: [function (data) {
      // 对 data 进行任意转换处理

      return data;
    }],

    // `headers` 是即将被发送的自定义请求头
    headers: {'X-Requested-With': 'XMLHttpRequest'},

    // `params` 是即将与请求一起发送的 URL 参数
    // 必须是一个无格式对象(plain object)或 URLSearchParams 对象
    params: {
      ID: 12345
    },

    // `paramsSerializer` 是一个负责 `params` 序列化的函数
    //              (e.g.              https://www.npmjs.com/package/qs,
http://api.jquery.com/jquery.param/)
    paramsSerializer: function(params) {
      return Qs.stringify(params, {arrayFormat: 'brackets'})
    },

    // `data` 是作为请求主体被发送的数据
    // 只适用于这些请求方法 'PUT', 'POST', 和 'PATCH'
    // 在没有设置 `transformRequest` 时，必须是以下类型之一：
    //   -   string,   plain   object,   ArrayBuffer,   ArrayBufferView,
URLSearchParams
    // - 浏览器专属：FormData, File, Blob
    // - Node 专属：  Stream
    data: {
```

```
    firstName: 'Fred'
  },

  // `timeout` 指定请求超时的毫秒数(0 表示无超时时间)
  // 如果请求话费了超过 `timeout` 的时间，请求将被中断
  timeout: 1000,

  // `withCredentials` 表示跨域请求时是否需要使用凭证
  withCredentials: false, // 默认的

  // `adapter` 允许自定义处理请求，以使测试更轻松
  // 返回一个 promise 并应用一个有效的响应 (查阅 [response docs](#response-api)).
  adapter: function (config) {
    /* ... */
  },

  // `auth` 表示应该使用 HTTP 基础验证，并提供凭据
  // 这将设置一个 `Authorization` 头，覆写掉现有的任意使用 `headers` 设置的自定义 `Authorization`头
  auth: {
    username: 'janedoe',
    password: 's00pers3cret'
  },

  // `responseType` 表示服务器响应的数据类型，可以是 'arraybuffer', 'blob', 'document', 'json', 'text', 'stream'
  responseType: 'json', // 默认的

  // `xsrfCookieName` 是用作 xsrf token 的值的cookie 的名称
  xsrfCookieName: 'XSRF-TOKEN', // default

  // `xsrfHeaderName` 是承载 xsrf token 的值的 HTTP 头的名称
  xsrfHeaderName: 'X-XSRF-TOKEN', // 默认的

  // `onUploadProgress` 允许为上传处理进度事件
  onUploadProgress: function (progressEvent) {
    // 对原生进度事件的处理
  },

  // `onDownloadProgress` 允许为下载处理进度事件
  onDownloadProgress: function (progressEvent) {
    // 对原生进度事件的处理
  },
```

```
    // `maxContentLength` 定义允许的响应内容的最大尺寸
    maxContentLength: 2000,

    // `validateStatus` 定义对于给定的 HTTP 响应状态码是 resolve 或 reject
promise 。如果 `validateStatus` 返回 `true` (或者设置为 `null` 或
`undefined`)，promise 将被 resolve; 否则，promise 将被 rejecte
    validateStatus: function (status) {
      return status >= 200 && status < 300; // 默认的
    },

    // `maxRedirects` 定义在 node.js 中 follow 的最大重定向数目
    // 如果设置为0，将不会 follow 任何重定向
    maxRedirects: 5, // 默认的

    // `httpAgent` 和 `httpsAgent` 分别在 node.js 中用于定义在执行 http 和
https 时使用的自定义代理。允许像这样配置选项:
    // `keepAlive` 默认没有启用
    httpAgent: new http.Agent({ keepAlive: true }),
    httpsAgent: new https.Agent({ keepAlive: true }),

    // 'proxy' 定义代理服务器的主机名称和端口
    // `auth` 表示 HTTP 基础验证应当用于连接代理，并提供凭据
    // 这将会设置一个 `Proxy-Authorization` 头，覆写掉已有的通过使用 `header`
设置的自定义 `Proxy-Authorization` 头。
    proxy: {
      host: '127.0.0.1',
      port: 9000,
      auth: : {
        username: 'mikeymike',
        password: 'rapunz31'
      }
    },

    // `cancelToken` 指定用于取消请求的 cancel token
    // (查看后面的 Cancellation 这节了解更多)
    cancelToken: new CancelToken(function (cancel) {
    })
  }
```

5.1.6 响应结构

某个请求的响应包含以下信息：

```
{
  // `data` 由服务器提供的响应
  data: {},

  // `status` 来自服务器响应的 HTTP 状态码
  status: 200,

  // `statusText` 来自服务器响应的 HTTP 状态信息
  statusText: 'OK',

  // `headers` 服务器响应的头
  headers: {},

  // `config` 是为请求提供的配置信息
  config: {}
}
```

使用 then 时，你将接收到下面这样的响应：

```
axios.get('/user/12345')
  .then(function(response) {
    console.log(response.data);
    console.log(response.status);
    console.log(response.statusText);
    console.log(response.headers);
    console.log(response.config);
  });
```

在使用 catch 时，或传递 rejection callback 作为 then 的第二个参数时，响应可以通过 error 对象被使用。

5.1.7　配置的默认值/defaults

你可以指定将被用在各个请求的配置默认值。

5.1.7.1　全局的 axios 默认值

```
axios.defaults.baseURL = 'https://api.example.com';
axios.defaults.headers.common['Authorization'] = AUTH_TOKEN;
axios.defaults.headers.post['Content-Type']   =   'application/x-www
-form-urlencoded';
```

5.1.7.2 自定义实例默认值

```
// 创建实例时设置配置的默认值
var instance = axios.create({
  baseURL: 'https://api.example.com'
});

// 在实例已创建后修改默认值
instance.defaults.headers.common['Authorization'] = AUTH_TOKEN;
```

5.1.7.3 配置的优先顺序

配置会以一个优先顺序进行合并。这个顺序是：在 lib/defaults.js 找到库的默认值，然后是实例的 defaults 属性，最后是请求的 config 参数。后者将优先于前者。这里是一个例子：

```
// 使用由库提供的配置的默认值来创建实例
// 此时超时配置的默认值是 `0`
var instance = axios.create();

// 覆写库的超时默认值
// 现在，在超时前，所有请求都会等待 2.5 秒
instance.defaults.timeout = 2500;

// 为已知需要花费很长时间的请求覆写超时设置
instance.get('/longRequest', {
  timeout: 5000
});
```

5.1.8 拦截器

在请求或响应被 then 或 catch 处理前拦截它们。

```
// 添加请求拦截器
axios.interceptors.request.use(function (config) {
    // 在发送请求之前做些什么
    return config;
  }, function (error) {
    // 对请求错误做些什么
    return Promise.reject(error);
  });
```

```
// 添加响应拦截器
axios.interceptors.response.use(function (response) {
    // 对响应数据做点什么
    return response;
  }, function (error) {
    // 对响应错误做点什么
    return Promise.reject(error);
  });
```

如果你想在稍后移除拦截器，可以这样做：

```
var myInterceptor = axios.interceptors.request.use(function () {/*...*/});
axios.interceptors.request.eject(myInterceptor);
```

可以为自定义 axios 实例添加拦截器：

```
var instance = axios.create();
instance.interceptors.request.use(function () {/*...*/});
```

5.1.9　错误处理

```
axios.get('/user/12345')
  .catch(function (error) {
    if (error.response) {
      // 请求已发出，但服务器响应的状态码不在 2xx 范围内
      console.log(error.response.data);
      console.log(error.response.status);
      console.log(error.response.headers);
    } else {
      // Something happened in setting up the request that triggered an Error
      console.log('Error', error.message);
    }
    console.log(error.config);
  });
```

可以使用 validateStatus 配置选项定义一个自定义 HTTP 状态码的错误范围。

```
axios.get('/user/12345', {
  validateStatus: function (status) {
    return status < 500; // 状态码在大于或等于 500 时才会 reject
  }
})
```

5.1.10 取消

使用 cancel token 取消请求。

Axios 的 cancel token API 基于 cancelable promises proposal，它还处于第一阶段。

可以使用 CancelToken.source 工厂方法创建 cancel token，方法如下：

```
var CancelToken = axios.CancelToken;
var source = CancelToken.source();

axios.get('/user/12345', {
  cancelToken: source.token
}).catch(function(thrown) {
  if (axios.isCancel(thrown)) {
    console.log('Request canceled', thrown.message);
  } else {
    // 处理错误
  }
});

// 取消请求（message 参数是可选的）
source.cancel('Operation canceled by the user.');
```

还可以通过传递一个 executor 函数到 CancelToken 的构造函数来创建 cancel token：

```
var CancelToken = axios.CancelToken;
var cancel;

axios.get('/user/12345', {
  cancelToken: new CancelToken(function executor(c) {
    // executor 函数接收一个 cancel 函数作为参数
    cancel = c;
  })
});

// 取消请求
cancel();
```

可以使用同一个 cancel token 取消多个请求。

5.1.11　Promises

Axios 依赖原生的 ES6 Promise 实现而被支持。

如果你的环境不支持 ES6 Promise，你可以使用 polyfill。

5.1.12　TypeScript

Axios 也支持了 TypeScript 的定义方式：

```
import axios from 'axios';
axios.get('/user?ID=12345');
```

5.2　Vuex 的使用

在熟悉完 Vuex 原理以后，介绍一下 Vuex 的使用。Vuex 的使用包括两部分，Vuex 的定义和 Vuex 的调用。

首先，通过一个简单案例来感受一下 Vuex 的定义调用方式。

```
const store = new Vuex.Store({
    state: {
        count: 0
    },
    mutations: {
        increment(state) {
            state.count++;
        }
    }
})
```

现在，我们可以通过 store.state 来获取状态对象，也可以通过 store.commit 方法来触发状态的修改：

```
store.commit('increment');
console.log(store.state.count)  // -> 1
```

接下来，从 Vuex 的核心概念中依次介绍使用方式。

5.2.1　State

Vue 组件获取 Vuex 里的状态如下。

```
const Counter = {
    template: '<div>{{ count }}</div>',
    computed: {
        count(){
            return store.state.count
        }
    }
}
```

这样每次 store.state.count 变化的时候，都会重新计算属性，触发更新关联 DOM。

我们可以在根组件里一次注入 store 选项，使子组件都可以使用这个 store，而不用每个组件都需要注入，实例如下：

```
const Counter = {
    template: '<div>{{ count }}</div>',
    computed: {
        count(){
            return this.$store.state.count
        }
    }
}

const app = new Vue({
    el: '#app',
    // 把 store 对象提供给"store"选项，这样可以把 store 的实例注入所有的子组件
    store,
    components: {Counter},
    template: '
        <div class="app">
            <counter></counter>
        </div>
    '
});
```

Vuex 提供了 mapState 的语法糖写法，目的就是让我们在书写上更简洁。下面是一个复杂案例的简化写法：

```
import { mapState } from 'Vuex'

export default {
    // ...
    computed: mapState({
```

```
        // 箭头函数可使代码更简练
        count: state => state.count,

        // 传字符串参数 'count' 等同于 `state => state.count`
        countAlias: 'count',

        // 为了能够使用 `this` 获取局部状态，必须使用常规函数
        countPlusLocalState (state) {
            return state.count + this.localCount
        }
    })
}
```

5.2.2　Getters

Getters 是对 state 的派生，因此在组件部分调用跟 state 没什么太大的区别，例如：

```
computed: {
    doneTodosCount(){
        return this.$store.getters.doneTodoCount
    }
}
```

重要的部分在 store 中 getters 的定义部分：

```
const store = new Vuex.Store({
    state: {
        todos: [
            { id: 1, text: '...', done: true },
            { id: 2, text: '...', done: false }
        ]
    },
    getters: {
        doneTodoCount: state => {
            return state.todos.filter(todo => todo.done)
        }
    }
});

// getters 也接受其他 getters 作为第二个参数
getters: {
    // ...
    doneTodoCount: (state, getters) => {
```

```
        return getters.doneTodos.length
    }
}
```

语法糖方面使用规则如下：

```
// ...
computed: {
    ...mapGetters({
        'doneTodoCount',
        // 如果你想换个名字，this.doneCount 为 store.getters.doneTodoCount
        doneCount: 'doneTodoCount'
    })
}
// ...
```

5.2.3 Mutations & Actions

mutationshe 和 Actions 的使用规则重点在 store 定义端，Vue 组件的使用则是简单的方法执行，示例如下：

```
const store = new Vuex.Store({
    state: {
        count: 1
    },
    mutations: {
        increment (state, payload) {
            // 变更状态
            state.count++
        }
    },
    actions: {
        increment (context, payload) {
            context.commit('increment')
        }
    }
})

// 使用
store.commit('increment');
store.dispatch('increment');
```

提交 commit 和 dispatch 可以追加一个 payload，而事实上，payload 可以是个

简单参数，同时也可以是一个对象，于是就有了如下的调用规则：

```
// ...
mutations: {
    increment (state, payload) {
        state.count += payload.amount
    }
}
actions: {
    increment (context, payload) {
        context.commit('increment', payload)
    }
}
// 使用
store.commit('increment', {
    amount: 10
})
store.dispatch('increment', {
    amount: 10
})
```

在 commit 和 dispatch 里还有一种对象风格的提交方式，直接使用包含 type 属性的对象如下：

```
store.commit({
    type: 'increment',
    amount: 10
});

store.dispatch({
    type: 'increment',
    amount: 10
});

mutations: {
    increment(state, payload){
        this.count += payload.amount;
    }
}
actions: {
    increment(context, payload){
        context.commit('increment', payload)
    }
}
```

mapMutations 和 mapActions 的使用规则跟之前的语法糖类似：

```
export default {
    // ...
    methods: {
        ...mapMutations([
          'increment' // 映射 this.increment() 为 this.$store.commit
('increment')
        ]),
        ...mapMutations({
          add: 'increment' // 映射 this.add() 为 this.$store.commit
('increment')
        }),
        ...mapActions([
          'increment' // 映射 this.increment() 为 this.$store.dispatch
('increment')
        ]),
        ...mapActions({
          add: 'increment' // 映射 this.add() 为 this.$store.dispatch
('increment')
        })
    }
}
```

因为 actions 是异步执行，因此存在一种组合 actions：

```
actions: {
    actionA ({ commit }) {
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                commit('someMutation')
                resolve()
            }, 1000)
        })
    },
    actionB ({ dispatch, commit }) {
        return dispatch('actionA').then(() => {
            commit('someOtherMutation')
        })
    }
}

//使用
store.dispatch('actionB');
```

```
// 关于嵌套还有一种新特性（async/await）写法
actions: {
    async actionA ({ commit }) {
        commit('gotData', await getData())
    },
    async actionB ({ dispatch, commit }) {
        await dispatch('actionA') // 等待 actionA 完成
        commit('gotOtherData', await getOtherData())
    }
}
```

5.2.4　Modules

modules 存在的意义是解决状态树臃肿问题，将状态树切割成几个子状态树，那么在使用上定义 modules，即是定义状态树（store），具体使用方法如下：

```
const moduleA = {
    state: { ... },
    mutations: { ... },
    actions: { ... },
    getters: { ... }
}

const moduleB = {
    state: { ... },
    mutations: { ... },
    actions: { ... }
}

const store = new Vuex.Store({
    modules: {
        a: moduleA,
        b: moduleB
    }
})

store.state.a // -> moduleA 的状态
store.state.b // -> moduleB 的状态
```

对于模块内部的 mutation 和 getter，接收的第一个参数是模块的局部状态对象；同样，对于模块内部的 action，局部状态通过 context.state 暴露出来，根节点状态则为 context.rootState；对于模块内部的 getter，根节点状态会作为第三个参

数暴露出来。

```
mutations: {
   increment (state) {
     // 这里的 `state` 对象是模块的局部状态
     state.count++
   }
 },

 getters: {
   sumWithRootCount (state, getters, rootState) {
     return state.count + rootState.count
   }
 },

 actions: {
   incrementIfOddOnRootSum ({ state, commit, rootState }) {
     if ((state.count + rootState.count) % 2 === 1) {
       commit('increment')
     }
   }
 }
```

既然是模块，就应该存在嵌套，而且每个模块都有对应的 action、mutation 和 getter。因此定义命名空间可以提高模块的可重用性，让改模块下的子模块依据当前的命名空间规范自动调整命名。例如：

```
const store = new Vuex.Store({
  modules: {
    account: {
        namespaced: true,
        // 模块内容 (module assets)
        state: { ... }, // 模块内的状态已经是嵌套的了，使用'namespace'属性不会对其产生影响
        getters: {
          isAdmin(){ ... } // -> getters('account/isAdmin')
        },
        modules: {
          myPage: {
              state: { ... },
              getters: {
                  profile(){ ... } //-> getters['account.profile']
              }
          },
```

```
        // 进一步嵌套命名空间
        posts: {
            namespaced: true,
            state: { ... },
            getters: {
                popular(){ ... } // -> getters['account/posts/ polular']
            }
        }
    }
  }
})
```

5.2.4.1　命名空间模块内访问全局内容

如果希望使用全局的 state 与 getter，rootState 和 rootGetter 会作为第三和第四个参数传入 getter，也会通过 context 对象的属性传入 action。

若需要在全局命名空间内分发 action 或提交 mutation，将{ root: true }作为第三个参数传给 dispatch 或 commit 即可。简答示例如下：

```
...
foo: {
    namespaced: true,
    getters: {
        someGetter(state, getters, rootState, rootGetters) {
            getters.someOtherGetter // -> 'foo/someOtherGetter'
            rootGetters.someOtherGetter // -> 'someOtherGetter'
        },
        someOtherGetter: state => {
            ...
        }
    },
    actions: {
        someAction({ dispatch, commit, getters, rootGetters }) {
            getters.someGetter // 'foo.someGetter'
            rootGetters.someGetter // 'someGetter'
            dispatch('someOtherAction') // 'foo.someOtherAction'
            dispatch('someOtherAction',   null,   {root:   true})   //
'someOtherAction'
            commit('someMutation') // 'foo.someMutation'
            commit('someMutation', null, {root: true}) // 'someMutation'
        },
        someOtherMutation(ctx, payload) { ... }
```

```
    }
}
...
```

5.2.4.2 带命名空间的绑定函数

一般的的语法糖写法如下：

```
computed: {
    ...mapState ({
        a: state => state.some.nested.module.a,
        b: state => state.some.nested.module.b
    })
},
methods: {
    ...mapActions ({
        'some/nested/module/foo',
        'some/nested/module/bar'
    })
}
```

针对上面的写法，有个优化的书写方式，提取公共的空间名称字符串作为第一个参数传递给上述函数，这样所有的绑定都是基于该模块的上下文，在理解和阅读上更简单。

```
computed: {
    ...mapState ('some/nested/module', {
        a: state => state.a,
        b: state => state.b
    })
},
methods: {
    ...mapActions ('some/nested/module', {
        'foo',
        'bar'
    })
}
```

5.2.4.3 给插件开发者的注意事项

如果你开发的插件提供了模块并允许用户将其添加到 Vuex store，可能需要考虑模块的空间命名问题。对于这种情况，你可以通过插件的参数对象来允许用户指定空间名称。

```
export function createPlugin (options = {}){
    return function (store) {
        const namespace = options.namespace || ''
        store.dispatch(namespace + 'pluginAction')
    }
}
```

5.2.4.4　模块动态注册

在 store 创建之后，我们可以通过 store.registerModule 方法注册模块：

```
store.registerModule('myModule', {
    // ...
})
store.registerModule(['nested', 'myModule'], {
    // ...
})
```

我们可以通过 store.state.myModule 和 store.state.nested.myModule 访问相应模块的状态。与此同时，你也可以通过 store.unregisterModule 来动态卸载模块，注意，此时不可用来卸载静态模块（只能卸载 register 的模块，不能卸载创建 store 时声明的模块）。

5.2.5　模块重用

模块重用的场景也会经常发生，例如：

- 多个 store 公共一个 module；
- 一个 store 里多次注册同一个 module。

我们将模块状态通过一个纯对象管理，那么这个状态对象会通过引用被共享，会导致数据污染问题。因此我们在书写这样一个对象时候通过一个函数来声明模块状态（跟解决 Vue 组件内 data 的数据污染解决方案一致）。示例如下：

```
const myModule = {
    state(){
        return {
            foo: 'bar'
        }
    },
    // mutation、action 和 getter 等~
}
```

Vuex 本身是进行状态管理，因此在使用上把握两个方向：状态定义和状态管理（读取、修改）。状态定义也就是定义全局 store，完成被管理状态的注册工作；而状态的使用发生在 Vue 组件里。同时 Vuex 提供了语法糖，更加方便我们使用 Vuex。最后还有模块的引用实现解构 store 的能力，解决 store 臃肿的问题。

5.3 Vue-router 使用

下面介绍 Vue-router 的使用。Vue-router 的使用包含两部分，router 的定义和调用。

5.3.1 安装

```
npm install Vue-router
```

如果在一个模块化工程中使用它，必须要通过 Vue.use()明确地安装路由功能：

```
import Vue from 'Vue'
import VueRouter from 'Vue-router'
Vue.use(VueRouter)
```

5.3.2 开始

5.3.2.1 HTML 调用方式

```
<script src="https://unpkg.com/Vue/dist/Vue.js"></script>
<script
src="https://unpkg.com/Vue-router/dist/Vue-router.js"></script>

<div id="app">
    <h1>Hello App!</h1>
    <p>
        <!-- 使用 router-link 组件来导航. -->
        <!-- 通过传入 `to` 属性指定链接. -->
        <!-- <router-link> 默认会被渲染成一个 `<a>` 标签 -->
        <router-link to="/foo">Go to Foo</router-link>
        <router-link to="/bar">Go to Bar</router-link>
    </p>
    <!-- 路由出口 -->
```

```
    <!-- 路由匹配到的组件将渲染在这里 -->
    <router-view></router-view>
</div>
```

5.3.2.2　JavaScript 调用方式

```
// 0. 如果使用模块化机制编程，导入 Vue 和 VueRouter，要调用 Vue.use(VueRouter)

  // 1. 定义（路由）组件。
  // 可以从其他文件 import 进来
  const Foo = { template: '<div>foo</div>' }
  const Bar = { template: '<div>bar</div>' }

  // 2. 定义路由
  // 每个路由应该映射一个组件。其中"component" 可以是
  // 通过 Vue.extend() 创建的组件构造器，
  // 或者，只是一个组件配置对象。
  // 我们晚点再讨论嵌套路由。
  const routes = [
    { path: '/foo', component: Foo },
    { path: '/bar', component: Bar }
  ]

  // 3. 创建 router 实例，然后传 `routes` 配置
  // 你还可以传别的配置参数，不过先这么简单着吧。
  const router = new VueRouter({
    routes // (缩写) 相当于 routes: routes
  })

  // 4. 创建和挂载根实例。
  // 记得要通过 router 配置参数注入路由，
  // 从而让整个应用都有路由功能
  const app = new Vue({
    router
  }).$mount('#app')
```

5.3.3　动态路由匹配

动态路由匹配指的是需要根据把某种模式的路由，全都映射到同一个组件。例如，我们需要根据用户 id 的不同，需要使用同一个 User 组件来进行渲染。可以通过使用 Vue-router 中的动态路径参数达到目的。

```
const User = {
    template : '<div>User</div>'
}
const router = new VueRouter({
    routes: [{
        //动态路径参数，以:开头
        {path: '/user/:id',component: User}
    }]
})
```

一个路径参数使用：标记。但匹配到一个路由时，参数值会被设置到this.$route.params，可以在每个组件内使用。我们可以更新 User 的模板，输出当前用户的 ID：

```
const User = {
  template: '<div>User {{ $route.params.id }}</div>'
}
```

在一个路由中可以设置多段路径参数，对应的值都会设置到$route.params 里。

```
<table>
    <thead>
        <tr>
            <th>模式</th>
            <th>匹配路径</th>
            <th>$route.params</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>/user/:username</td>
            <td>/user/evan</td>
            <td>{ username: 'evan' }</td>
        </tr>
        <tr>
            <td>/user/:username/post/:post_id</td>
            <td>/user/evan/post/123</td>
            <td>{ username: 'evan', post_id: 123 }</td>
        </tr>
    </tbody>
</table>
```

5.3.3.1　响应路由参数的变化

当使用路由参数时，例如从/user/foo 到/user/bar，原来的组件实例会被复用。因为两个路由都渲染同一个组件，比起销毁再创建，复用则显得更加高效。不过，这意味着组件的生命周期钩子不会再被调用。

复用组件时，想对路由参数做出响应的话，你可以简单地 watch（监测变化）$route 对象。

```
const User = {
  template: '...',
  watch: {
    '$route' (to, from) {
      // 对路由变化作出响应...
    }
  }
}
```

5.3.3.2　高级匹配模式

Vue-router 使用 path-to-regexp 作为路径匹配引擎，所以支持很多高级的匹配模式，例如：可选的动态路径参数、匹配零个或多个、一个或多个，甚至是自定义正则匹配。

5.3.3.3　匹配优先级

有时候，同一个路径可以匹配多个路由，此时，匹配的优先级就按照路由的定义顺序：谁先定义的，谁的优先级就最高。##嵌套路由借助 Vue-router，使用嵌套路由配置，可以表达 URL 中各段动态路径也按某种结构对应嵌套的各层组件。接着上节的例子：

```
<div id="app">
  <router-view></router-view>
</div>
<script type="text/javascript">
   const User = {
     template: '<div>User {{ $route.params.id }}</div>'
   }
```

```
    const router = new VueRouter({
      routes: [
        { path: '/user/:id', component: User }
      ]
    })
</script>
```

一个渲染组件同样可以包含自己的嵌套<router-view>，例如，在 User 组件的模板添加<router-view>：

```
const User = {
    template: `
      <div class="user">
        <h2>User {{ $route.params.id }}</h2>
        <router-view></router-view>
      </div>
    `
  }
```

要在嵌套的出口中渲染组件，需要在 VueRouter 的参数中使用 children 配置：

```
const router = new VueRouter({
  routes: [
    { path: '/user/:id', component: User,
      children: [
        {
          // 当 /user/:id/profile 匹配成功，
          // UserProfile 会被渲染在 User 的 <router-view> 中
          path: 'profile',
          component: UserProfile
        },
        {
          // 当 /user/:id/posts 匹配成功
          // UserPosts 会被渲染在 User 的 <router-view> 中
          path: 'posts',
          component: UserPosts
        }
      ]
    }
  ]
})
```

要注意，以 / 开头的嵌套路径会被当做根路径。这让你充分地使用嵌套组件

而无须设置嵌套的路径。此时，基于上面的配置，当你访问/user/foo 时，User 的出口是不会渲染任何东西的，这是因为没有匹配到合适的子路由。如果你想要渲染点什么，可以提供一个空的子路由：

```
const router = new VueRouter({
  routes: [
    {
      path: '/user/:id', component: User,
      children: [
        // 当 /user/:id 匹配成功，
        // UserHome 会被渲染在 User 的 <router-view> 中
        { path: '', component: UserHome },

        // ...其他子路由
      ]
    }
  ]
})
```

5.3.4　编程式导航

我们可以通过<router-link>标签定义导航链接，我们还可以借助router的方法，通过编写代码来实现 router.push(location)，这个方法会向 history 栈中添加一个新记录，所以，当用户点击浏览器回退按钮时，则回到之前的 URL。

声明式：

<router-link :to="...">

编程式：

router.push(...)

该方法的参数可以是一个字符串路径，或者是一个描述地址的对象。例如：

```
// 字符串
router.push('home')

// 对象
router.push({ path: 'home' })

// 命名的路由
router.push({ name: 'user', params: { userId: 123 }})
```

```
// 带查询参数，变成 /register?plan=private
router.push({ path: 'register', query: { plan: 'private' }})

router.replace(location)
```

跟 router.push 很像，唯一的不同就是，它不会向 history 添加新记录，而是跟它的方法名一样——替换掉当前的 history 记录。

声明式：

<router-link :to="..." replace>

编程式：

router.replace(...)

router.go(n)

这个方法的参数是一个整数，意思是在 history 记录中向前或者后退多少步，类似 window.history.go(n)。

5.3.5 命名路由

有时候，通过一个名称来标识一个路由显得更方便一些，特别是在链接一个路由，或者是执行一些跳转的时候。你可以在创建 Router 实例的时候，在 routes 配置中给某个路由设置名称。

```
const router = new VueRouter({
  routes: [
    {
      path: '/user/:userId',
      name: 'user',
      component: User
    }
  ]
})
```

要链接到一个命名路由，可以给 router-link 的 to 属性传一个对象：

```
<router-link :to="{ name: 'user', params: { userId: 123 }}">User
</router-link>
```

等同于执行

```
router.push({ name: 'user', params: { userId: 123 }})
```

这两种方式都会把路由导航到/user/123 路径中。

5.3.6　命名视图

同级展示多个视图。可以在界面中拥有多个单独命名的视图，而不是只有一个单独的出口。如果 router-view 没有设置名字，那么默认为 default。

```
<router-view class="view one"></router-view>
<router-view class="view two" name="a"></router-view>
<router-view class="view three" name="b"></router-view>
```

一个视图使用一个组件渲染，因此对于同个路由，多个视图就需要多个组件。确保正确使用 components 配置（带上 s）：

```
const router = new VueRouter({
  routes: [
    {
      path: '/',
      components: {
        default: Foo,
        a: Bar,
        b: Baz
      }
    }
  ]
})
```

5.3.7　重定向和别名

5.3.7.1　重定向

重定向的意思是，当用户访问/a 时，URL 将会被替换成/b，然后匹配路由为/b。重定向也是通过 routes 配置来完成，下面例子是从/a 重定向到/b：

```
const router = new VueRouter({
  routes: [
    { path: '/a', redirect: '/b' }
  ]
})
```

重定向的目标也可以是一个命名的路由：

```
const router = new VueRouter({
  routes: [
```

```
    { path: '/a', redirect: { name: 'foo' }}
  ]
})
```

甚至是一个方法，动态返回重定向目标：

```
const router = new VueRouter({
  routes: [
    { path: '/a', redirect: to => {
      // 方法接收 目标路由 作为参数
      // return 重定向的 字符串路径/路径对象
    }}
  ]
})
```

5.3.7.2 别名

别名的意思是，/a 的别名是/b，意味着，当用户访问/b 时，URL 会保持为/b，但是路由匹配则为/a，就像用户访问/a 一样。

```
const router = new VueRouter({
  routes: [
    { path: '/a', component: A, alias: '/b' }
  ]
})
```

别名的功能让你可以自由地将 UI 结构映射到任意的 URL，而不是受限于配置的嵌套路由结构。

5.3.8 HTML5 History 模式

Vue-router 默认 hash 模式——使用 URL 的 hash 来模拟一个完整的 URL，于是当 URL 改变时，页面不会重新加载。如果不想要很丑的 hash，我们可以用路由的 history 模式，这种模式充分利用 history.pushState API 来完成 URL 跳转而无须重新加载页面。

```
const router = new VueRouter({
  mode: 'history',
  routes: [...]
})
```

当你使用 history 模式时，URL 就像正常的 url，例如 http://yoursite.com/user/id，

也好看！

不过这种模式要玩好，还需要后台配置支持。因为我们的应用是个单页客户端应用，如果后台没有正确的配置，当用户在浏览器直接访问 http://oursite.com/user/id 就会返回 404，这就不好看了。

所以呢，你要在服务端增加一个覆盖所有情况的候选资源：如果 URL 匹配不到任何静态资源，则应该返回同一个 index.html 页面，这个页面就是你的 App 依赖的页面。

5.3.9　后端配置例子

5.3.9.1　Apache

```
<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteBase /
  RewriteRule ^index\.html$ - [L]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteRule . /index.html [L]
</IfModule>
```

5.3.9.2　nginx

```
location / {
  try_files $uri $uri/ /index.html;
}
```

5.3.9.3　Node.js (Express)

<a>https://github.com/bripkens/connect-history-api-fallback

5.3.10　警告

给个警告，因为这么做以后，你的服务器就不再返回 404 错误页面，因为对于所有路径都会返回 index.html 文件。为了避免这种情况，你应该在 Vue 应用里面覆盖所有的路由情况，然后再给出一个 404 页面。

```
const router = new VueRouter({
```

```
    mode: 'history',
    routes: [
      { path: '*', component: NotFoundComponent }
    ]
  })
```

或者，如果你是用 Node.js 作后台，可以使用服务端的路由来匹配 URL，当没有匹配到路由的时候返回 404，从而实现 fallback。

5.3.11 导航钩子

Vue-router 提供的导航钩子主要用来拦截导航，让它完成跳转或取消。有多种方式可以在路由导航发生时执行钩子：全局的、单个路由独享的，或者组件级的。

5.3.11.1 全局钩子

你可以使用 router.beforeEach 注册一个全局的 before 钩子。

```
const router = new VueRouter({ ... })

router.beforeEach((to, from, next) => {
  // ...
})
```

当一个导航触发时，全局的 before 钩子按照创建顺序调用。钩子是异步解析执行，此时导航在所有钩子 resolve 完之前一直处于等待中。每个钩子方法接收三个参数：

```
to: Route: 即将要进入的目标 路由对象
from: Route: 当前导航正要离开的路由
next: Function: 一定要调用该方法来 resolve 这个钩子。执行效果依赖 next 方法的调用参数。
    next(): 进行管道中的下一个钩子。如果全部钩子执行完了，则导航的状态就是 confirmed （确认的）。
    next(false): 中断当前的导航。如果浏览器的 URL 改变了（可能是用户手动或者浏览器后退按钮），那么 URL 地址会重置到 from 路由对应的地址。
    next('/') 或者 next({ path: '/' }): 跳转到一个不同的地址。当前的导航被中断，然后进行一个新的导航。
```

确保要调用 next 方法，否则钩子就不会被 resolved。同样可以注册一个全局的 after 钩子，不过它不像 before 钩子那样，after 钩子没有 next 方法，不能改变导航。

```
router.afterEach(route => {
  // ...
})
```

5.3.11.2　某个路由独享的钩子

你可以在路由配置上直接定义 beforeEnter 钩子：

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      beforeEnter: (to, from, next) => {
        // ...
      }
    }
  ]
})
```

这些钩子与全局 before 钩子的方法参数是一样的。

5.3.11.3　组件内的钩子

你可以在路由组件内直接定义以下路由导航钩子：

beforeRouteEnter beforeRouteUpdate (2.2 新增) beforeRouteLeave

```
const Foo = {
  template: `...`,
  beforeRouteEnter (to, from, next) {
    // 在渲染该组件的对应路由被 confirm 前调用
    // 不！能！获取组件实例 `this`
    // 因为当钩子执行前，组件实例还没被创建
  },
  beforeRouteUpdate (to, from, next) {
    // 在当前路由改变，但是该组件被复用时调用
    // 举例来说，对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和 /foo/2 之间跳转的时候，
    // 由于会渲染同样的 Foo 组件，因此组件实例会被复用。而这个钩子就会在这个情况下被调用。
    // 可以访问组件实例 `this`
  },
  beforeRouteLeave (to, from, next) {
```

```
    // 导航离开该组件的对应路由时调用
    // 可以访问组件实例 `this`
  }
}
```

beforeRouteEnter 钩子不能访问 this，因为钩子在导航确认前被调用，因此即将登场的新组件还没被创建。

不过，你可以通过传一个回调给 next 来访问组件实例。在导航被确认的时候执行回调，并且把组件实例作为回调方法的参数。

```
beforeRouteEnter (to, from, next) {
  next(vm => {
    // 通过 `vm` 访问组件实例
  })
}
```

你可以在 beforeRouteLeave 中直接访问 this。这个 leave 钩子通常用来禁止用户在还未保存修改前突然离开。可以通过 next(false)来取消导航。##路由元信息，定义路由的时候可以配置 meta 字段：

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      children: [
        {
          path: 'bar',
          component: Bar,
          // a meta field
          meta: { requiresAuth: true }
        }
      ]
    }
  ]
})
```

首先，我们称呼 routes 配置中的每个路由对象为路由记录。路由记录可以是嵌套的，因此，当一个路由匹配成功后，它可能匹配多个路由记录。

例如，根据上面的路由配置，/foo/bar 这个 URL 将会匹配父路由记录以及子路由记录。

一个路由匹配到的所有路由记录会暴露为$route 对象（还有在导航钩子中的 route 对象）的$route.matched 数组。因此，我们需要遍历$route.matched 来检查路由记录中的 meta 字段。

下面例子展示在全局导航钩子中检查 meta 字段：

```
router.beforeEach((to, from, next) => {
  if (to.matched.some(record => record.meta.requiresAuth)) {
    // this route requires auth, check if logged in
    // if not, redirect to login page.
    if (!auth.loggedIn()) {
      next({
        path: '/login',
        query: { redirect: to.fullPath }
      })
    } else {
      next()
    }
  } else {
    next() // 确保一定要调用 next()
  }
})
```

5.3.12　过渡动效

<router-view>是基本的动态组件，所以我们可以用<transition>组件给它添加一些过渡效果：

```
<transition>
  <router-view></router-view>
</transition>
```

<transition> 的所有功能在这里同样适用。

5.3.12.1　单个路由的过渡

上面的用法会给所有路由设置一样的过渡效果，如果你想让每个路由组件有各自的过渡效果，可以在各路由组件内使用<transition>并设置不同的 name。

```
const Foo = {
  template: `
    <transition name="slide">
      <div class="foo">...</div>
```

```
      </transition>
    `
  }

  const Bar = {
    template: `
      <transition name="fade">
        <div class="bar">...</div>
      </transition>
    `
  }
```

5.3.12.2 基于路由的动态过渡

还可以基于当前路由与目标路由的变化关系，动态设置过渡效果：

```
<!-- 使用动态的 transition name -->
<transition :name="transitionName">
  <router-view></router-view>
</transition>

<script type="text/javascript">
// 接着在父组件内
// watch $route 决定使用哪种过渡

......

watch: {
  '$route' (to, from) {
    const toDepth = to.path.split('/').length
    const fromDepth = from.path.split('/').length
    this.transitionName = toDepth < fromDepth ? 'slide-right' :
'slide-left'
  }
}
</script>
```

5.3.13 数据获取

有时候，进入某个路由后，需要从服务器获取数据。例如，在渲染用户信息时，你需要从服务器获取用户的数据。我们可以通过两种方式来实现。

导航完成之后获取：先完成导航，然后在接下来的组件生命周期钩子中获取

数据。在数据获取期间显示『加载中』之类的指示。

导航完成之前获取：导航完成前，在路由的 enter 钩子中获取数据，在数据获取成功后执行导航。

5.3.13.1　导航完成后获取数据

当你使用这种方式时，我们会马上导航和渲染组件，然后在组件的 created 钩子中获取数据。这让我们有机会在数据获取期间展示一下 loading 状态，还可以在不同视图间展示不同的 loading 状态。

假设我们有一个 Post 组件，需要基于$route.params.id 获取文章数据：

```
<template>
  <div class="post">
    <div class="loading" v-if="loading">
      Loading...
    </div>

    <div v-if="error" class="error">
      {{ error }}
    </div>

    <div v-if="post" class="content">
      <h2>{{ post.title }}</h2>
      <p>{{ post.body }}</p>
    </div>
  </div>
</template>
<script type="text/javascript">
  export default {
    data () {
      return {
        loading: false,
        post: null,
        error: null
      }
    },
    created () {
      // 组件创建完后获取数据，
      // 此时 data 已经被 observed 了
      this.fetchData()
    },
    watch: {
```

```
        // 如果路由有变化，会再次执行该方法
        '$route': 'fetchData'
      },
      methods: {
        fetchData () {
          this.error = this.post = null
          this.loading = true
          // replace getPost with your data fetching util / API wrapper
          getPost(this.$route.params.id, (err, post) => {
            this.loading = false
            if (err) {
              this.error = err.toString()
            } else {
              this.post = post
            }
          })
        }
      }
    }
  </script>
```

5.3.13.2 在导航完成前获取数据

通过这种方式，我们在导航转入新的路由前获取数据。我们可以在接下来的组件 beforeRouteEnter 钩子中获取数据，当数据获取成功后只调用 next 方法。

```
  export default {
    data () {
      return {
        post: null,
        error: null
      }
    },
    beforeRouteEnter (to, from, next) {
      getPost(to.params.id, (err, post) =>
        if (err) {
          // display some global error message
          next(false)
        } else {
          next(vm => {
            vm.post = post
          })
        }
```

```
    })
  },
  // 路由改变前，组件就已经渲染完了
  // 逻辑稍稍不同
  watch: {
    $route () {
      this.post = null
      getPost(this.$route.params.id, (err, post) => {
        if (err) {
          this.error = err.toString()
        } else {
          this.post = post
        }
      })
    }
  }
}
```

在为后面的视图获取数据时，用户会停留在当前的界面，因此建议在数据获取期间，显示一些进度条或者别的指示。如果数据获取失败，同样有必要展示一些全局的错误提醒。

5.3.14　滚动行为

使用前端路由，当切换到新路由时，想要页面滚到顶部，或者是保持原先的滚动位置，就像重新加载页面那样。Vue-router 能做到，而且更好，它让你可以自定义路由切换时页面如何滚动。

```
<strong>注意：这个功能只在 HTML5 history 模式下可用。</strong>
```

当创建一个 Router 实例，你可以提供一个 scrollBehavior 方法。

```
const router = new VueRouter({
  routes: [...],
  scrollBehavior (to, from, savedPosition) {
    // return 期望滚动到哪个位置
  }
})
```

scrollBehavior 方法接收 to 和 from 路由对象。第三个参数 savedPosition 当且仅当 popstate 导航（通过浏览器的前进/后退按钮触发）时才可用。

这个方法返回滚动位置的对象信息，具体如下。

```
{ x: number, y: number }

{ selector: string }
```

如果返回一个布尔假的值，或者是一个空对象，那么不会发生滚动。

举例：

```
scrollBehavior (to, from, savedPosition) {
  return { x: 0, y: 0 }
}
```

对于所有路由导航，简单地让页面滚动到顶部。

返回 savedPosition，在按下后退/前进按钮时，就会像浏览器的原生表现那样：

```
scrollBehavior (to, from, savedPosition) {
  if (savedPosition) {
    return savedPosition
  } else {
    return { x: 0, y: 0 }
  }
}
```

如果你要模拟『滚动到锚点』的行为：

```
scrollBehavior (to, from, savedPosition) {
  if (to.hash) {
    return {
      selector: to.hash
    }
  }
}
```

我们还可以利用路由元信息 更细颗粒度地控制滚动。

5.3.15 懒加载

当打包构建应用时，Javascript 包会变得非常大，影响页面加载。如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应组件，这样就更加高效了。

结合 Vue 的异步组件和 Webpack 的 code splitting feature，轻松实现路由组件的懒加载。

我们要做的就是把路由对应的组件定义成异步组件：

```
const Foo = resolve => {
  // require.ensure 是 Webpack 的特殊语法，用来设置 code-split point
  // （代码分块）
  require.ensure(['./Foo.vue'], () => {
    resolve(require('./Foo.vue'))
  })
}
```

这里还有另一种代码分块的语法，使用 AMD 风格的 require，于是就更简单了：

```
const Foo = resolve => require(['./Foo.vue'], resolve)
```

不需要改变任何路由配置，跟之前一样使用 Foo：

```
const router = new VueRouter({
  routes: [
    { path: '/foo', component: Foo }
  ]
})
```

把组件按组分块。

有时候我们想把某个路由下的所有组件都打包在同个异步 chunk 中。只需要给 chunk 命名，提供 require.ensure 第三个参数作为 chunk 的名称：

```
const Foo = r => require.ensure([], () => r(require('./Foo.vue')),
'group-foo')
const Bar = r => require.ensure([], () => r(require('./Bar.vue')),
'group-foo')
const Baz = r => require.ensure([], () => r(require('./Baz.vue')),
'group-foo')
```

Webpack 将相同 chunk 下的所有异步模块打包到一个异步块里面——这也意味着我们无须明确列出 require.ensure 的依赖(传空数组就行)。

本章介绍 vue-router 的使用，主要针对 vue-router 的配置进行了详细解释。包括 Router 的配置，router-link 和 router-view 等的各项参数的使用等。

读 者 圈

第 6 章　Vue 项目优化

6.1　状态过渡

6.1.1　过渡的概念

Vue 只有在插入、更新或者移除 dom 元素的时候才会产生过渡效果，过渡效果可以通过不同的方式实现，在官方文档中有如下几种：

- 在 CSS 过渡和动画中自动应用 class；
- 可以配合使用第三方 CSS 动画库，如 Animate.css；
- 在过渡钩子函数中使用 JavaScript 直接操作 DOM；
- 可以配合使用第三方 JavaScript 动画库，如 Velocity.js。

而实际上，上述的四种方式实际上就是两种，一种是利用 CSS 过渡或者动画，另一个是利用 js 中的钩子函数。

6.1.2　CSS 过渡

通常所常用的过渡是 CSS 过渡。

```
<div id="transition-example">
  <button @click="show = !show">
    Hello Vue
  </button>
  <transition name="slide-fade">
    <p v-if="show">Hello Vue</p>
  </transition>
</div>
new Vue({
  el: '#example-1',
  data: {
    show: true
  }
```

```
})
/* 可以设置不同的进入和离开动画 */
/* 设置持续时间和动画函数 */
.slide-fade-enter-active {
  transition: all .3s ease;
}
.slide-fade-leave-active {
  transition: all .8s cubic-bezier(1.0, 0.5, 0.8, 1.0);
}
.slide-fade-enter, .slide-fade-leave-to
/* .slide-fade-leave-active for below version 2.1.8 */ {
  transform: translateX(10px);
  opacity: 0;
}
```

在这个例子中，我们可以像 CSS3 中的动画属性那样，设置不同方法进入或离开动画。同样，可以很方便地设计持续的时间、运动的状态以及动画函数，具有如下四个属性：

v 是在没有 transition name 的时候调用，v-enter，v-enter-active，v-leave，v-leave-active。

<img width="763" alt="wx20170820-204314 2x" src=
"https://user-images.githubusercontent.com/6932025/29494990-47fca028-85e8-11e7-8465-1968ce03de6c.png">

6.1.3　Javascript 钩子

同样的，我们可以在属性中声明 javascript 钩子。举例如下：

```
<transition
 v-on:before-enter="beforeEnter"
 v-on:enter="enter"
 v-on:after-enter="afterEnter"
 v-on:enter-cancelled="enterCancelled"
 v-on:before-leave="beforeLeave"
 v-on:leave="leave"
 v-on:after-leave="afterLeave"
v-on:leave-cancelled="leaveCancelled">
</transition>
methods: {
  // --------
  // 进入中
```

```
    // --------
    beforeEnter: function (el) {
      // ...
    },
    // 此回调函数是可选项的设置
    // 与 CSS 结合时使用
    enter: function (el, done) {
      // ...
      done()
    },
    afterEnter: function (el) {
      // ...
    },
    enterCancelled: function (el) {
      // ...
    },
    // --------
    // 离开时
    // --------
    beforeLeave: function (el) {
      // ...
    },
    // 此回调函数是可选项的设置
    // 与 CSS 结合时使用
    leave: function (el, done) {
      // ...
      done()
    },
    afterLeave: function (el) {
      // ...
    },
    // leaveCancelled 只用于 v-show 中
    leaveCancelled: function (el) {
      // ...
    }
  }methods: {
    // --------
    // 进入中
    // --------
    beforeEnter: function (el) {
      // ...
    },
    // 此回调函数是可选项的设置
    // 与 CSS 结合时使用
```

```
    enter: function (el, done) {
      // ...
      done()
    },
    afterEnter: function (el) {
      // ...
    },
    enterCancelled: function (el) {
      // ...
    },
    // --------
    // 离开时
    // --------
    beforeLeave: function (el) {
      // ...
    },
    // 此回调函数是可选项的设置
    // 与 CSS 结合时使用
    leave: function (el, done) {
      // ...
      done()
    },
    afterLeave: function (el) {
      // ...
    },
    // leaveCancelled 只用于 v-show 中
    leaveCancelled: function (el) {
      // ...
    }
  }
```

这些钩子函数可以结合 CSS transitions/animations 使用，也可以单独使用。

值得注意的是，当只用 JavaScript 过渡的时候，在 enter 和 leave 中，回调函数 done 是必须的。否则，它们会被同步调用，过渡会立即完成。

transition 的加入可以更好地适配更多的使用场景，而这里仅介绍了一下 transition 的基本用法，更多的实例炫酷用法这里就不再赘述了。你可以参照 Vue 官网中的一些例子来更好地演练一下 Vue 的 transition 的使用方式。而在实际的使用上就是前文所提到的 CSS 以及 js 的钩子这两种方式。

6.2 Vue 项目的自动化测试

说到自动化测试，许多开发团队都是听说过、尝试过的，但最后都止步于尝试，不能将 TDD（测试驱动开发）、BDD（行为驱动开发）的完整流程贯彻到项目中。思考其中的原因：终究还是收益抵不上成本。

很多后端开发人员可能写过很多自动化的单元测试代码，但是对前端测试一头雾水。这是因为相对于后端开发人员的自动化单元测试，前端的自动化测试成本更高。

自动化测试就是通过自动化脚本将一个又一个测试用例串起来，每个测试用例都要模拟环境、模拟输入，然后断言输出。前端自动化最难的地方就是模拟环境、模拟输入和断言输出了。我们可以试想一下现实中的使用场景。

模拟环境：首先前端代码是跑在不同的终端环境上的，纯粹地使用某台机子的运行环境进行模拟是无法发现真正存在的问题。所以我们的测试用例必须运行在真实的环境下，这里面包括不同的机器：Android、ios、pc、macbook；不同的系统：window10、window8、linux、mac；不同的运行载体：IE、safari、chrome、firefox、Opera、Android webview、UIWebview、WKWebview；不同的网络环境：WiFi、4G、3G、offline。

模拟输入：前端的输入不好模拟，在 PC 上有鼠标 click，double Click、drag、mouseDown、mouseOver、input 等，在 mobile 上有 swipe、tap、scroll、摇一摇、屏幕翻转等。相对于后端的单元测试，前端的输入种类繁多，每一种模拟起来都十分复杂，而且很多 bug 隐藏在几种连贯的输入之后才会复现。

断言输出：前端的断言不是简单的判断值是否相等，很多情况是即使值相等、效果完全不一样。很多展示效果更是不能通过简单的断言来检测，比如区域是否能滑动，输入时键盘是否正确弹起等。

当你跨越千山万水把上面的问题解决了，测试用例写好了，功能代码写好了，完美！然后 UE 跑过来和你说这根线往左边移动一像素的时候，你会瞬间崩溃。可能因为这一像素你很多测试用例都得重写。所以前端自动化测试的成本真不一定抵得上收益。

但是困难不代表解决不了，部分场景不适合不代表所有场景都不适合！

正因为面临这么多的困难，我们的前端社区开发出了很多工具帮我们解决这些问题。本章节主要是结合 Vue 这个框架介绍前端自动化测试的一些工具和方法。

我们使用 Vue-cli 去新建一个 Vue 的新项目，在这个项目中开启默认的 unit tests 和 e2e tests。

```
bogon:work xiaorenhui$ vue init webpack vueExample

? Project name vue-example
? Project description A Vue.js project
? Author kukuv <kukuv>
? Vue build standalone
? Install vue-router? No
? Use ESLint to lint your code? No
? Setup unit tests with Karma + Mocha? Yes
? Setup e2e tests with Nightwatch? Yes
```

下面列举下这个新项目中涉及到的一些开源项目。

- karma。

Karma 是一个基于 Node.js 的 JavaScript 测试执行过程管理工具（Test Runner）。该工具可用于测试所有主流 Web 浏览器，也可集成到 CI（Continuous integration）工具，也可和其他代码编辑器一起使用。这个测试工具的一个强大特性就是它可以监控（Watch）文件的变化，然后自行执行，通过 console.log 显示测试结果。

- Mocha。

mocha 是一款功能丰富的 javascript 单元测试框架，它既可以运行在 nodejs 环境中，也可以运行在浏览器环境中。

- Nightwatch。

Nightwatch 是一套基于 Node.js 的测试框架，使用 Selenium WebDriver API 以将 Web 应用测试自动化。它提供了简单的语法，支持使用 JavaScript 和 CSS 选择器，来编写运行在 Selenium 服务器上的端到端测试。

- phantomjs。

一个基于 webkit 内核的无头浏览器，即没有 UI 界面，即它就是一个浏览器，只是其内的点击、翻页等人为相关操作需要程序设计实现。

- sinon-chai。

sinon-chai 是 sinon 和 chai 这两个断言库的结合，提供丰富的断言方法。

很多人看到这么多新名词一定头晕，心想一个单元测试怎么需要懂这么多东西。而实际情况是上面只是单元测试框架的一小部分，还有许多框架没有列出来。正因为前端的自动化测试面临着许多问题，所以我们才有这么多的框架来帮忙解决问题。

6.2.1 unit tests

我们先来分析一下这个项目中的 unit tests，这里面用到了 Karma、Mocha、sinon-chai、phantomjs。项目中已经有一个默认的单元测试例子。karma 作为测试执行过程管理工具把 Mocha、sinon-chai、phantomjs 等框架组织起来。Mocha 用来描述测试用例、sinon-chai 用来断言，然后使用 phamtomjs 作为运行环境来跑测试用例。

npm install 将依赖的库都安装好，这里面 phantomjs 的依赖会比较难装，如果你之间没有安装过 phantom，因为 phantom 比较大，而且加上国内的网络环境等原因。如果 phantomjs 装不上可以尝试使用 chrome 作为运行环境，这需要安装 "karma-chrome-launcher"，需要修改配置文件。

然后 npm run unit 运行一下 unit tests，如果提示权限问题，就使用 sudo 来提升一下权限。运行之后我们看一下如下的目录结构。

```
└── unit
    ├── coverage  代码覆盖率报告，src 下面的 index.html 可以直接用浏览器打开
    │   ├── lcov-report
    │   │   ├── base.css
    │   │   ├── index.html
    │   │   ├── prettify.css
    │   │   ├── prettify.js
    │   │   ├── sort-arrow-sprite.png
    │   │   ├── sorter.js
    │   │   └── src
    │   │       ├── App.vue.html
    │   │       ├── components
    │   │       │   ├── Hello.vue.html
    │   │       │   └── index.html
    │   │       └── index.html
    │   └── lcov.info
    ├── index.js 运行测试用例前先加载的文件，方便统计代码覆盖率
    ├── karma.conf.js karma 的配置文件
    └── specs 所有的测试用例都放在这里
        └── Hello.spec.js
```

```
// 加载所有的测试用例、 testsContext.keys().forEach(testsContext)这种写法是 webpack 中的加载目录下所有文件的写法

const testsContext = require.context('./specs', true, /\.spec$/)
testsContext.keys().forEach(testsContext)
```

```
  // 加载所有代码文件，方便统计代码覆盖率
  const srcContext = require.context('../../src', true, /^\.\/
(?!main(\.js)?$)/)
  srcContext.keys().forEach(srcContext)
  config.set({
    // 在几个环境里跑你的测试用例
    // browsers: ['PhantomJS','Chrome'],
    browsers: ['Chrome'],
    // 默认加载几个框架
    frameworks: ['mocha', 'sinon-chai', 'phantomjs-shim'],
    // 使用那些汇报框架
    reporters: ['spec', 'coverage'],
    // 预加载文件
    files: ['./index.js'],
    // 预处理
    preprocessors: {
      './index.js': ['webpack', 'sourcemap']
    },
    // webpack 配置
    webpack: webpackConfig,
    webpackMiddleware: {
      noInfo: true
    },
    // coverage 配置
    coverageReporter: {
      dir: './coverage',
      reporters: [
        { type: 'lcov', subdir: '.' },
        { type: 'text-summary' }
      ]
    }
  })
```

上面使用的插件，例如 mocha、spec、coverage，除了 karma 默认自带的以外，都需要你在 npm 上安装对应的插件，例如：

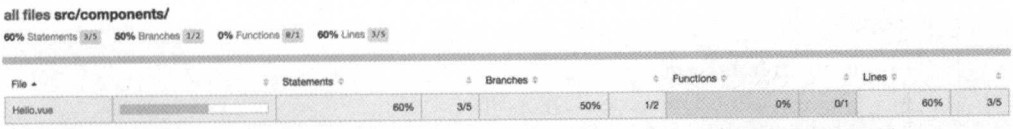
```
  "karma": "^1.4.1",
    "karma-chrome-launcher": "^2.2.0",
    "karma-coverage": "^1.1.1",
    "karma-mocha": "^1.3.0",
    "karma-phantomjs-launcher": "^1.0.2",
    "karma-phantomjs-shim": "^1.4.0",
    "karma-sinon-chai": "^1.3.1",
    "karma-sourcemap-loader": "^0.3.7",
```

```
      "karma-spec-reporter": "0.0.31",
      "karma-webpack": "^2.0.2",
   > vue-exampl@1.0.0 unit /Users/xiaorenhui/work/vueExample
   >  cross-env  BABEL_ENV=test  karma  start  test/unit/karma.conf.js
--single-run

   07 09 2017 12:08:13.004:INFO [karma]: Karma v1.7.0 server started at
http://0.0.0.0:9876/
   07 09 2017 12:08:13.007:INFO [launcher]: Launching browser Chrome with
unlimited concurrency
   07 09 2017 12:08:13.015:INFO [launcher]: Starting browser Chrome
   07 09 2017 12:08:15.475:INFO [Chrome 60.0.3112 (Mac OS X 10.12.3)]:
Connected on socket qDaxr51TuQCfQBcVAAAA with id 73077049
   INFO LOG: 'Download the Vue Devtools extension for a better development
experience:
   https://github.com/vuejs/vue-devtools'
   LOG LOG: 'data'

    Hello.vue
      ✓ should render correct contents

   Chrome 60.0.3112 (Mac OS X 10.12.3): Executed 1 of 1 SUCCESS (0.024 secs
/ 0.011 secs)
   TOTAL: 1 SUCCESS


   ================================          Coverage          summary
================================
   Statements   : 60% ( 3/5 )
   Branches     : 50% ( 1/2 )
   Functions    : 0% ( 0/1 )
   Lines        : 60% ( 3/5 )
   ================================================================================
=============
```

我修改了一下 Hello.vue 这个组件，可以看到 coverage 里精确地显示了测试代码的覆盖率，下面是我做的修改：

```
   export default {
     name: 'hello',
     data () {
        console.log('data');
        function aa() {
```

```
    }
    if(false){
        console.log('data aa');
    }
  return {
    msg: 'Welcome to Your Vue.js App'
  }
 },
 methods:{
    aa(){
        console.log('methods aa');
    }
 }
}
```



打开 reporter 下面的 index.html，我们可以看到代码覆盖的具体情况。点开 Hello.vue 更有直观的方式展示哪些代码被覆盖了，哪些没有。

```
<script>
export default {
  name: 'hello',
  data () {
1×    console.log('data');
      function aa() {

      }
1×    if(false){
        console.log('data aa');
      }
1×   return {
       msg: 'Welcome to Your Vue.js App'
     }
  },
  methods:{
     aa(){
        console.log('methods aa');
     }
  }
}
</script>
```

6.2.2　e2e 测试

既然我们已经有了单元测试，那 e2e 测试和单元测试有什么区别呢？

Nightwatch 是前端 e2e 测试的一个有代表性的框架。单元测试 TDD 的粒度很细，我们会为许多函数、方法去写单元测试，而 e2e 更接近 BDD。直白地说，就是 TDD 的测试单元是一个个函数、方法，而 BDD 测试的单元是一个个预期的行为表现。e2e 做的事情就是打开浏览器，并且真正访问我们最终的页面，然后，在这个真实的浏览器、真实的页面中去做各种断言，而单元测试不会要去我们去访问最终的页面，单元测试要保证的是一个个单元是没有问题的，但这些单元组合起来跑在页面上是否有问题，不是单元测试能够保证的，尤其是在前端这种模拟环境、模拟输入非常复杂的领域中，这是单元测试的短板，而 e2e 测试就是用来解决这些短板的。

我们来看看项目中使用 Nightwatch 来进行 e2e 测试的例子。

首先看以下目录：

```
├── e2e
│   ├── custom-assertions
│   │   └── elementCount.js 自定义的断言方法
│   ├── nightwatch.conf.js nightwatch 的配置文件
│   ├── reports
│   │   ├── CHROME_60.0.3112.101_Mac\ OS\ X_test.xml
│   │   └── CHROME_60.0.3112.113_Mac\ OS\ X_test.xml
│   ├── runner.js  bootstrap 文件，起我们的页面 server 和 nightwatch 文件
│   └── specs
│       └── test.js 测试用例
```

其次看单元测试关系如图 6-1 所示。

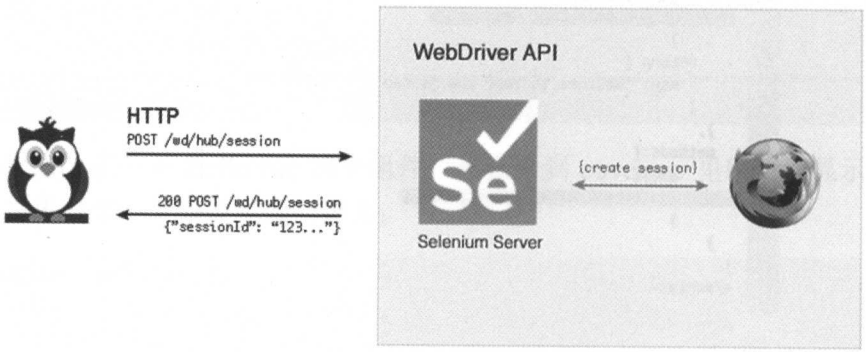


图 6-1

selenium 是一个用 java 写的 e2e 测试工具集，它的 API 被纳入 w3c 的 webDriver

API 中，nightWatch 是对 selenium 的一个 nodejs 封装。所有我们需要在配置文件中配置 selenium。

```
  src_folders: ['test/e2e/specs'],
    output_folder: 'test/e2e/reports',
    custom_assertions_path: ['test/e2e/custom-assertions'],
     // 对 selenium 的配置
    selenium: {
      start_process: true,
      server_path: require('selenium-server').path,
      host: '127.0.0.1',
      port: 4444,
      cli_args: {
        'webdriver.chrome.driver': require('chromedriver').path
      }
    },
     // 测试环境的配置
    test_settings: {
      default: {
        selenium_port: 4444,
        selenium_host: 'localhost',
        silent: true,
        globals: {
          devServerURL:  'http://localhost:'  +  (process.env.PORT  ||
config.dev.port)
        }
      },

      chrome: {
        desiredCapabilities: {
          browserName: 'chrome',
          javascriptEnabled: true,
          acceptSslCerts: true
        }
      },

      firefox: {
        desiredCapabilities: {
          browserName: 'firefox',
          javascriptEnabled: true,
          acceptSslCerts: true
        }
      }
    }
```

下面的 runner 需要先起我们的一个网页服务，然后，再起 nightWatch 服务。

```
var server = require('../../build/dev-server.js')

server.ready.then(() => {
  // 2. run the nightwatch test suite against it
  // to run in additional browsers:
  //      1. add an entry in test/e2e/nightwatch.conf.json under
"test_settings"
  //    2. add it to the --env flag below
  // or override the environment flag, for example: `npm run e2e -- --env
chrome,firefox`
  // For more information on Nightwatch's config file, see
  // http://nightwatchjs.org/guide#settings-file
  var opts = process.argv.slice(2)
    console.log(opts);
  if (opts.indexOf('--config') === -1) {
    opts = opts.concat(['--config', 'test/e2e/nightwatch.conf.js'])
  }
  if (opts.indexOf('--env') === -1) {
    opts = opts.concat(['--env', 'chrome,firefox'])
  }

  var spawn = require('cross-spawn')
  var runner = spawn('./node_modules/.bin/nightwatch', opts, { stdio:
'inherit' })

  runner.on('exit', function (code) {
    server.close()
    process.exit(code)
  })

  runner.on('error', function (err) {
    server.close()
    throw err
  })
})
```

sudo npm run e2e 后

```
> node test/e2e/runner.js

> Starting dev server...
```

```
Starting to optimize CSS...
> Listening at http://localhost:8080

[]
Starting selenium server... started - PID:  74459

[Test] Test Suite
=====================

Running:  default e2e tests
 ✔ Element <#app> was visible after 81 milliseconds.
 ✔ Testing if element <.hello> is present.
 ✔ Testing if element <h1> contains text: "Welcome to Your Vue.js App".
 ✔ Testing if element <img> has count: 1

OK. 4 assertions passed. (3.951s)
```

在控制台上我们能看到各种断言的结果。

6.3　Typescript Support

6.3.1　Typescript

TypeScript 是一种由微软开发的自由和开源的编程语言。它是 JavaScript 的一个严格超集，并添加了可选的静态类型和基于类型的面向对象编程。C#的首席架构师以及 Delphi 和 Turbo Pascal 的创始人安德斯·海尔斯伯格参与了 TypeScript 的开发。

TypeScript 设计目标是开发大型应用，然后转译成 JavaScript，编译出来的 Javascript 可以运行在任何浏览器上。Typescript 编译工具可以运行在任何服务器和系统上。由于 TypeScript 是 JavaScript 的严格超集，任何现有的 JavaScript 程序都是合法的 TypeScript 程序。

TypeScript 支持为现存 JavaScript 库添加类型信息的定义文件，方便其他程序，像使用静态类型的值一样使用现有库中的值。目前有第三方提供常用库如 jQuery、MongoDB、Node.js 和 D3.js 的定义文件。

TypeScript 编译器本身也是用 TypeScript 写成的，并被转译为 JavaScript，以 Apache License 2 发布。

Typescript 是开源的。

6.3.2 安装 Typescript

一般有两种方式获得 Typescript 工具。

- 通过 npm（node.js 的包管理工具）；
- 安装 TypeScript 的 Visual Studio 插件。

```
本节默认使用的是 npm(node.js 的包管理工具)进行安装
默认大家都已经成功安装好了 node 环境以及 npm 包管理器
```

下面我们使用 npm 安装 Typescript

```
npm install -g typescript
```

TypeScript 最大的优势之一便是增强了编辑器和 IDE 的功能，包括代码补全、接口提示、跳转到定义、重构等。

主流的编辑器都支持 TypeScript，这里推荐使用 Visual Studio Code。

它是一款开源，跨终端的轻量级编辑器，内置了 TypeScript 支持。

另外它本身也是用 TypeScript 编写的。

下载安装：https://code.visualstudio.com/

当然你也可以下载其他编辑器或 IDE 对 TypeScript 的支持，这是目前主流可以支持 Typescript 的：

- Sublime Text；
- Atom；
- WebStorm；
- Vim；
- Emacs；
- Eclipse；
- Visual Studio 2015；
- Visual Studio 2013。

6.3.3 Typescript 和 Vue 结合

6.3.3.1 新建项目

我们先来创建一个新的项目：

```
mkdir typescript-vue-project
cd typescript-vue-project
```

接下来，看一下我们项目的基础架构：

```
typescript-vue-project/
├─ dist/
└─ src/
   └─ components/
```

这里主要的 Typescript 文件将放在 components 文件夹下，然后通过 Typescript 编译器，进行 webpack 最后生成 dist。webpack 最后会生成 dist 目录，并运行在生产环境上。

6.3.3.2　初始化项目

我们先初始化这个项目：

```
npm init
```

配置上我们都是选择默认，一路回车之后，最终生成 package.json 文件。

6.3.3.3　安装项目依赖

```
npm install –save-dev typescript webpack ts-loader css-loader vue-loader
vue-template-compiler@2.2.1
```

6.3.3.4　在根目录创建 Typescript 配置文件

```
{
   "compilerOptions": {
       "outDir": "./built/",
       "sourceMap": true,
       "strict": true,
       "module": "es2015",
       "moduleResolution": "node",
       "target": "es5"
    },
    "include": [
       "./src/**/*"
    ]
}
```

需要注意的是，这里面需要将 strict 设置为 true。开启 Typescript 的严格模式可以给我们带来更好的体验。

6.3.3.5 添加 webpack 的配置

我们需要创建一个 webpack.conf.js 来进行对 Typescript 的编译。

```
var path = require('path')
var webpack = require('webpack')

module.exports = {
  entry: './src/index.ts',
  output: {
    path: path.resolve(__dirname, './dist'),
    publicPath: '/dist/',
    filename: 'build.js'
  },
  module: {
    rules: [
      {
        test: /\.vue$/,
        loader: 'vue-loader',
        options: {
          loaders: {
            // Since sass-loader (weirdly) has SCSS as its default parse
mode, we map
            // the "scss" and "sass" values for the lang attribute to the
right configs here.
            // other preprocessors should work out of the box, no loader
config like this necessary.
            'scss': 'vue-style-loader!css-loader!sass-loader',
            'sass':
'vue-style-loader!css-loader!sass-loader?indentedSyntax',
          }
          // other vue-loader options go here
        }
      },
      {
        test: /\.tsx?$/,
        loader: 'ts-loader',
        exclude: /node_modules/,
        options: {
          appendTsSuffixTo: [/\.vue$/],
        }
```

```
      },
      {
        test: /\.(png|jpg|gif|svg)$/,
        loader: 'file-loader',
        options: {
          name: '[name].[ext]?[hash]'
        }
      }
    ]
  },
  resolve: {
    extensions: ['.ts', '.js', '.vue', '.json'],
    alias: {
      'vue$': 'vue/dist/vue.esm.js'
    }
  },
  devServer: {
    historyApiFallback: true,
    noInfo: true
  },
  performance: {
    hints: false
  },
  devtool: '#eval-source-map'
}

if (process.env.NODE_ENV === 'production') {
  module.exports.devtool = '#source-map'
  // http://vue-loader.vuejs.org/en/workflow/production.html
  module.exports.plugins = (module.exports.plugins || []).concat([
    new webpack.DefinePlugin({
      'process.env': {
        NODE_ENV: '"production"'
      }
    }),
    new webpack.optimize.UglifyJsPlugin({
      sourceMap: true,
      compress: {
        warnings: false
      }
    }),
    new webpack.LoaderOptionsPlugin({
      minimize: true
    })
```

```
    ])
}
```

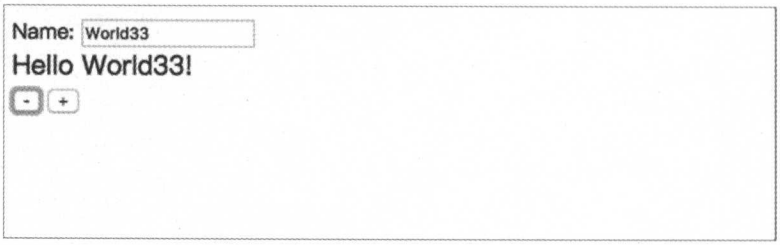
6.3.3.6 创建构建脚本

打开 package.json，在 scripts 那里配置 build 指令

```
"script": {
      "build": "webpack ",
      "test": "echo \"Error: no test specified\" && exit 1"
  }
```

一旦我们创建了入口，我们就可以通过 npm build 来进行项目构建。同时可以在运行的时候，同步构建。

```
npm run build -- --watch
```



6.3.3.7 创建一个基本的 ts 文件

下面让我们创建一个最简单的 Typescript&vue 示例在./src/index.ts 上：

```
// index.ts
import Vue from "vue";
import HelloComponent from "./components/Hello.vue";

let v = new Vue({
    el: "#app",
    template: `
    <div>
        Name: <input v-model="name" type="text">
        <hello-component :name="name" :initialEnthusiasm="5" />
    </div>
    `,
    data: { name: "World" },
```

```
    components: {
        HelloComponent
    }
});
```

然后在根目录下创建一个 index.html 文件，现在运行 npm run build 在浏览器中打开文件，这个时候你就会看到 hello world。

不错，我们已经成功地将 Typescript 和 Vue 结合起来了。

6.3.3.8　添加组件

正如你刚刚所做到的，利用 Vue 我们构建出了一个非常简单的界面。并且只需要一些数据就能够操作页面上的显示，而对于一些复杂的业务，Vue 同样是非常灵活的，可以将应用拆分成不同的组件。

```
<!-- src/components/Hello.vue -->
<template>
    <div>
        <div class="greeting">Hello {{name}}{{exclamationMarks}}</div>
        <button @click="decrement">-</button>
        <button @click="increment">+</button>
    </div>
</template>

<script lang="ts">
import Vue from "vue";

export default Vue.extend({
    props: ['name', 'initialEnthusiasm'],
    data() {
        return {
            enthusiasm: this.initialEnthusiasm,
        }
    },
    methods: {
        increment() { this.enthusiasm++; },
        decrement() {
            if (this.enthusiasm > 1) {
                this.enthusiasm--;
            }
        },
    },
    computed: {
```

```
        exclamationMarks(): string {
            return Array(this.enthusiasm + 1).join('!');
        }
    }
});
</script>

<style>
.greeting {
    font-size: 20px;
}
</style>
```

6.3.3.9 单文件组件

上面，我们在 src 下创建了 components 文件夹，并单独独立出来一个组件。我们在前面安装了 vue-loader 模块，并且在 webpack.conf.js 中指定了 `ts-loader` 的选项，这样就可以从单文件组件中提取出 Typescript 代码了。

并且，在单文件中，我们需要创建一个 Vue-shims.d.ts 文件。

```
declare module "*.vue" {
    import Vue from "vue";
    export default Vue;
}
```

我们不需要在任何地方引用这个文件，Typescript 语法会将这个文件自动包含进去。

在单文件组件中，我们需要注意一下：

- 在组件中<script lang='ts'></script> 才能使用 Typescript；
- 可以将<style>标签汇总的 CSS 分离到组件中，但是在.ts 组件中无法做到。

6.4 MPA

MPA（Multi-Page Application），多页面应用。我们通常在开发的时候都是单页面应用，即一个页面实例化一个 Vue 对象完成所有的页面逻辑。但是，有些应用场景里面需要切换到多页面应用，即一个模块，一个页面，一个 Vue 实例。多页面应用开发跟原始的多页面开发类似，不同功能在不同的页面里，跳转页面完成页面刷新加载内容。

关于 MPA 在 Vue 里有一些问题需要探讨，例如多页面的文件结构、公共资源、切换规则及优劣势等。

6.4.1　关于 MPA 的优劣势

说起 MPA 就不得不提到 SPA（Single-Page Application），我们在开发中无非就是从这两个方向里选择，那么 MPA 相对于 SPA 有哪些优势？更加适用于哪些场景呢？

MPA 的优势明显：

- 开发难度减小，相对于单页面应用，所有的逻辑关系都放在一个页面里，多页面开发将逻辑颗粒化，这样每个模块开发难度降低很多；
- 数据流更为简单，因为进行了颗粒化，所有的模块之间逻辑不存在数据共享，数据互相限制，因此数据流逻辑上更为简单；
- 不会因为业务复杂，某一个模块出现错误导致整个业务不能使用，而单页面会出现这样的问题。

MPA 的劣势也很明显：

- 多页面切换会影响用户体验；
- 多页面基于一些规定的配置，很多的多页面应用都是基于 webpack 规则制定，这增加了开发成本；
- 多页面应用也会增加冗余代码。

6.4.2　如何实现 MPA

接下来通过结合案例代码实现一个多页面应用，由于当前代码案例里是 SPA，重构成 MPA 代价较大，而且可能会影响其他部分关联代码的问题，因此我们会通过介绍一个 MPA 案例来帮助大家理解 MPA 的实现机制。具体的项目地址：vue-cli-multi-page。

6.4.2.1　目录结构

首先观察一下目录结构：

```
vue-cli-multi-page
    build
    config
    dist
        static
```

```
        views
    node_modules
    src
        assets
        components
        views
            home //一级目录
                list //二级目录
                    list.html
                    list.js
                    listApp.vue
            iconfont
            router
            tools
            vuxDemo
    test
```

通过上面目录结构重点了解部分模块的作用。

- build：构建项目的一些文件，其中很重要的一部分是对于多页面应用基于 webpack 的配置问题，这里不做重点分析，有兴趣的读者可以在地址链接里一探究竟，其最终目的是为了让我们独立开发的各个模块的单页面输出到一个文件下，并且依照文件路径实现切换调用，更多的感觉是一个规则定义者。
- dist：通过 webpack 的配置文件，最终开发源文件输出到这个文件夹下，static 下是静态资源而 views 则是每个页面，最终输出在浏览器里。
- src：开发资源文件夹，即多页面开发的主要工作区，从目录结构里我们可以明确看出，views 里每个文件夹都是一个独立的页面。拿 home 举例，默认一个列表文件夹，list.html 是一个没有内容的 html 外壳，而 list.js 完成将 listApp.vue 组件挂载到 list.html 壳子里（实例化一个 Vue 对象）。

```
// 大致展示每个文件的源码，以便更明确其功能

// list.html
<html>
    <head>
        ...
    </head>
    <body>
        // id 提供挂载点
        <div id="app"></div>
```

```
    </body>
</html>

// list.js
import Vue from 'vue'
import App from './listApp'

new Vue({
  render: h => h(App)
}).$mount('#app')

// listApp.vue 则是具体的 Vue 组件
```

每一个文件夹一个页面，实例化一个 Vue 对象。增加二级目录是方便页面很多的状况下便于归纳，同时所有的模块下都要是二级目录。接下来如何实现页面切换？共享数据问题？

6.4.2.2　公共资源 Lib.js

在多页面应用里会有一些公共资源被共享，本例子里通过 assets 里的 Lib.js 实现全局组件调用，实例如下：

```
// Lib.js 囊括了需要公用的所有能力

// css
require('common.css');
...
// plugin
Vue.use('plugin')
...
import X from 'X'
...
export default{
    ...
}

// *.vue 调用，实现资源共享
import lib from 'Lib'
```

Lib.js 抽离出公共资源进行初始化，然后在每个模块 import 里面即可使用。关于公共模块的界定在 webpack 也有实现的方案：在文件路径/build/webpack.prod.conf.js 里查找 minChunks，这个值决定了哪些资源可以被加载到公

共模块里，例如我们规定被四个页面引用了的资源认为是公共资源，提前加载，这个时候我们设置 minChunk 为 4。这个参数可以根据实际的业务项目进行弹性调整。

6.4.2.3 页面跳转

关于页面跳转，基于 webpack 的配置直接指定目的文件路径即可，这一部分配置在 webpack 配置文件里，由于这里只是介绍 MPA，关于 webpack 的具体配置不做具体介绍，有想了解的读者搜看源码即可。跳转规则定义如下：

```
    ...
     <group>
       <cell  title=" 多 页 面 路 由 "  value=""  is-link  link="../router/
details.html"></cell>
    </group>
    ...
    <group>
      <cell title="iconfont 图标展示" value="" is-link link="../iconfont/
list.html"></cell>
    </group>
    ...
```

如此配置完成页面切换，类似于 a 标签的路径配置效果。

6.5 Vue 的异构

组件化是至上而下的，一旦一个页面从某个 Dom 节点开始逐渐话之后，相当于从这个 Dom 节点开始所有的子辈节点都被组件化框架所接管了，这是因为 Vue 或 React 这类组件化框架都有一个重要的特性：那就是“数据=》视图”的一一对应关系，所以在被组件化框架所接管的区域内，所以的 Dom 操作都应该有组件化框架来完成，如果中间有其他的 Dom 操作，比如 jQuery 修改 Dom 节点元素，会破坏“数据=》视图”这个一一对应关系。

但是事实上，我们有大量的网页系统和框架并不是基于组件化框架所搭建的，所以很多时候我们必须拿出异构的方案。这里的异构指的是组件化框架和其他框架共存的情况。

我们主要介绍 Vue 异构方法，实际上 Vue 提供的 computed、watch 、directive 等对异构提供了很大的帮助。所以 Vue 的异构相对其他组件化框架来说更加容易

理解和操作。

组件化异构的核心思想就是在不破坏“数据=》视图”的前提下，将非组件化的代码封装成类组件化的代码。

6.5.1　不属于异构的情况

在列举 Vue 的异构方法之前，我们要分清楚什么情况才称之为异构，并不是引入了任意 js 库就叫异构。只有引入其他 js 库，并且我们使用该库去操作 Vue 所接管的 Dom 节点时才称之为异构。

下面是一个不是异构的例子。

我们从 github 上下载 Vue 的源代码，里面有一些实用的例子在 example 文件夹里，其中有一个《elastic-header》的例子，我们能看到这个例子里引用了 dynamic.js 这个 js 库来实现平滑的动画效果。那这里面算不算异构呢，需不需要我们采取什么特殊处理呢？会不会破坏“数据=》视图”的映射关系呢？其实是不会的。

```
85          },
86          stopDrag: function () {
87            if (this.dragging) {
88              this.dragging = false
89              dynamics.animate(this.c, {
90                x: 160,
91                y: 160
92              }, {
93                type: dynamics.spring,
94                duration: 700,
95                friction: 280
96              })
97            }
98          }
99        }
```

我们看这里面使用 dynamics.js 的方法，dynamics 被当成一个工具库来使用，并没有直接操作 Dom，而是改变 Vue 里面对象的数值，最终是由 Vue 去操作 Dom 的，所以并不会破坏“数据=》视图”的映射关系，所以我们并不需要担心引入这类的库与 Vue 混用会产生问题。

下面我们对 Vue 的异构方法做一些分类，不同的异构需求关注的重心也不太一样。

6.5.2 通过封装成 Vue 组件的方式实现异构

在 Vue 的 example 库里就有一个 Vue 异构的例子，在 example/select2 这个目录下。这个例子是 Vue 与 jQuery 的插件 select2 进行异构的例子，并且是通过将 select2 封装成 Vue 组件的方式实现异构的。

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Vue.js wrapper component example (jquery plugin:
select2)</title>
    <!-- Delete ".min" for console warnings in development -->
    <script src="../../dist/vue.min.js"></script>
    <script src="https://unpkg.com/jquery"></script>
    <script src="https://unpkg.com/select2@4.0.3"></script>
    <link
href="https://unpkg.com/select2@4.0.3/dist/css/select2.min.css"
rel="stylesheet">
    <style>
      html, body {
        font: 13px/18px sans-serif;
      }
      select {
        min-width: 300px;
      }
    </style>
  </head>
  <body>

    <div id="el">
    </div>

    <!-- using string template here to work around HTML <option> placement
restriction -->
    <script type="text/x-template" id="demo-template">
      <div>
        <p>Selected: {{ selected }}</p>
        <select2 :options="options" v-model="selected">
          <option disabled value="0">Select one</option>
        </select2>
      </div>
    </script>
```

```
<script type="text/x-template" id="select2-template">
  <select>
    <slot></slot>
  </select>
</script>

<script>
Vue.component('select2', {
  props: ['options', 'value'],
  template: '#select2-template',
  mounted: function () {
    var vm = this
    $(this.$el)
      .val(this.value)
      // init select2
      .select2({ data: this.options })
      // emit event on change.
      .on('change', function () {
        vm.$emit('input', this.value)
      })
  },
  watch: {
    value: function (value) {
      // update value
      $(this.$el).val(value).trigger('change')
    },
    options: function (options) {
      // update options
      $(this.$el).select2({ data: options })
    }
  },
  destroyed: function () {
    $(this.$el).off().select2('destroy')
  }
})

var vm = new Vue({
  el: '#el',
  template: '#demo-template',
  data: {
    selected: 1,
    options: [
      { id: 1, text: 'Hello' },
```

```
            { id: 2, text: 'World' }
          ]
        }
      })
      </script>
    </body>
  </html>
```

这里面最关键的代码就是组件的 mounted 方法和 watch 方法，异构最关键的事情有三件：

- Vue 要初始化其他库（组件化异构的时候一般在 mounted 方法里，因为这个时候能拿到 Dom 元素）；
- 当 Vue 数据发生变化的时候要调用其他库的方法更新页面；
- 当其他库的数据方法发生变化的时候要对 Vue 中的数据重新赋值，保证“数据=》视图”这个对应关系不变

```
  $(this.$el)
          .val(this.value)
          // 在 mounted 方法里初始化 select2 库
          .select2({ data: this.options })
          // 当其他库的数据方法发生变化的时候要对 Vue 中的数据重新赋值，保证“数据=》视图”这个对应关系不变
          .on('change', function () {
            vm.$emit('input', this.value)
          })
  watch: {
    // 当 Vue 数据发生变化的时候要调用其他库的方法更新页面
      value: function (value) {
        // update value
        $(this.$el).val(value).trigger('change')
      },
      options: function (options) {
        // update options
        $(this.$el).select2({ data: options })
      }
    },
```

6.5.3 通过 directive 的方式实现异构

下面我们将官方提供的例子修改一下，改成由 directive 的方式来实现异构。因为指令在 Vue2 里，能力已经被极大的削弱，但也依然具备能够实现异构的能力。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Vue.js wrapper component example (jquery plugin:
select2)</title>
    <!-- Delete ".min" for console warnings in development -->
    <script src="../../dist/vue.min.js"></script>
    <script src="https://unpkg.com/jquery"></script>
    <script src="https://unpkg.com/select2@4.0.3"></script>
    <link
href="https://unpkg.com/select2@4.0.3/dist/css/select2.min.css"
          rel="stylesheet">
    <style>
        html, oubody {
            font: 13px/18px sans-serif;
        }

        select {
            min-width: 300px;
        }
    </style>
</head>
<body>

<div id="el"></div>

<!-- using string template here to work around HTML <option> placement
restriction -->
<script type="text/x-template"
        id="demo-template">
    <div>
        <p>Selected: {{ selected }}</p>
    // 使用 v-select 指令
        <select   v-select="{options:options,onChange:onChange,value:
selected}"></select>
    </div>
</script>

<script>
    Vue.directive('select', {
        // 当绑定元素插入到 DOM 中。
```

```
        bind: function (el, value) {
            console.log('bind ' + arguments);
        },
        inserted: function (el, binding) {
            var value = binding.value
            // 聚焦元素
            $(el)
                    .val(value.value)
                    // init select2
                    .select2({data: value.options})
                    // emit event on change.
                    .on('select2:select', function (evt) {
                        var id = evt.params.data.id
                        value.onChange(id)
                    })
            console.log('inserted ' + arguments);
        },
        update: function (el, binding) {
            var val = $(el).val()
            var value = binding.value.value
            if (val !== value) {
                $(el).val(value).trigger('change')
            }
        },
        componentUpdated: function () {
            console.log('componentUpdated ' + arguments);
        },
        unbind: function () {
            console.log('unbind ' + arguments);
        }
    })

    var vm = new Vue({
        el: '#el',
        template: '#demo-template',
        data: {
            selected: 1,
            options: [
                {
                    id: 1,
                    text: 'Hello'
                },
                {
```

```
                id: 2,
                text: 'World'
            }
        ]
    },
    methods: {
        onChange: function (val) {
            debugger;
            this.selected = val
        }
    }
})
</script>
</body>
</html>
```

因为指令无法直接获取到 vm，所以我们通过 binding 里传入 onChange 的方法来更改组件的值。

这里面最关键的代码就是指令的 inserted 方法和 update 方法：

- 指令要初始化其他库（在 inserted 方法里，因为这时候能获取到对应的 Dom 节点）；
- 当 Vue 数据发生变化的时候要调用其他库的方法更新页面（在 update 方法里调用 select2 库的更新方法）；
- 当其他库的数据方法发生变化的时候要对 Vue 中的数据重新赋值，保证“数据=》视图”这个对应关系不变（注册事件监听，调用指令的 binding.value 值里的 onChange 方法）。

6.5.4　循环嵌套 Vue 组件

想象中很复杂的异构，其实只要把数据到视图的逻辑都理清楚了，实现起来非常容易。

我们再想象一个应用场景，如果这时候需要在 select2 这个 jQuery 插件库中再插入 Vue 组件应该怎么处理呢？

首先在由其他库中生成的 Dom 节点中插入 Vue 组件，只能通过 new Vue() 方法去重新初始化 Vue 组件。但是需要注意的是组件是一个可以多实例的概念，所以我们不能简单地给嵌套 Vue 组件一个 Id，因为当多实例的情况下通过这个 Id 能取到不只一个 Dom 节点，所以我们最佳方式是通过 Dom 节点树向下寻找，直

到找到能拿来初始化的对应元素，可以通过 jQuery 的 find 方法，也能通过生成 uuid 的方式来生成唯一的 Id，通过这些方式来保证组件在多实例的情况下，运行起来也不会出现问题。

6.6 服务端渲染

6.6.1 服务端渲染的概念

Vue.js 是构建客户端应用程序的框架。默认情况下，可以在浏览器中输出 Vue 组件，进行生成并操作 DOM。然而，也可以将同一个组件渲染为服务器端的 HTML 字符串，将它们直接发送到浏览器，最后将静态标记“混合”为客户端上完全交互的应用程序。

服务器渲染的 Vue.js 应用程序也可以被认为是“同构”或“通用”，因为应用程序的大部分代码都可以在服务器和客户端上运行。

6.6.2 用 Vue-ssr 的意义

- 前端用的是 Vue，后端渲染用 Vue-ssr，可以无缝地和前端连接起来；
- 使用 Vue-ssr 可以把数据渲染成 HTML，并在首屏展示，用户体验好，传统的前端 Vue，服务器第一次请求只返回#app 的空 DOM，当 js 和 ajax 请求完成，才会展示，体验差；
- 利用 SEO。

6.6.3 Vue-ssr 的作用

当然不是，它的最最主要作用是首屏渲染，其他都是次要的。比如有 3 个 tab 页签，只有第一个页签是首屏展示的，其他两个是通过点击才展示数据，那这样就没有必要把另外两个页签的数据也取出来，做 Vue-ssr，这样会增加服务器端的压力和流量。

6.6.4 Vue-ssr 学习难度

Vue-ssr 的学习，说实话，不是很容易，虽然现在网上的例子很多，官方也有一个例子 Vue-hackernews，但是官方给出的例子太复杂，属于大而全的例子，不适合什么概念的新手，网上的例子一般都是半个流程，比如只告诉你渲染简单的

模板，根本不会把项目中用到的整个流程都串起来，而且机理性的文章也少，增加了学习的难度。而本节内容就是让你理解并使用 Vue 的 ssr。

6.6.5　技术栈

Vue2+webpack2+Vuex+axios

6.6.6　前后端数据策略

在官方的例子当中，App 是带了 Vuex 跟 Vue-router 的，所以，preFetch 方案整个集成在这些库当中。从实验看，内部嵌套的 preFetch 是不会被调用的，只能从路由开始，同时中间要用到 Promise.all 的合并请求。

但是我觉得，Vue 的全家桶都不是必须的，而 Vuex 是应该必须有的。为什么这么说呢？刚开始上项目时，我也没打算用 Vuex，因为感觉那玩意没啥用，太复杂。后来一边做，一边就发现一个比较难解决的问题：兄弟组件间通信的问题！所以，但凡有这种组件之间通信问题的时候，建议还是加上 Vuex，可以使项目事半功倍。

好吧，我觉得这是一个相当简单粗暴的获取数据的办法，但其实也很难解耦，不然就要从路由直接推算数据才行。

6.6.7　性能影响

```
module.exports={render:function(){with(this) {
  return _h('li', {
    staticClass: "news-item"
  }, [_h('span', {
    staticClass: "score"
  }, [_s(item.score)]), " ", _h('span', {
    staticClass: "title"
  }, [(item.url) ? [_h('a', {
    attrs: {
      "href": item.url,
      "target": "_blank"
    }
  }, [_s(item.title)]), " ", _h('span', {
    staticClass: "host"
  }, ["(" + _s(_f("host")(item.url)) + ")"])] : [_h('router-link', {
    attrs: {
```

```
      "to": '/item/' + item.id
    }
```

编译后大致还能看到 Virtual DOM 的影子，会有一些性能开销。但是速度上应该没有模板引擎快的。

6.6.8 安装

```
npm install Vue Vue-server-renderer --save
```

注意：

- 推荐使用 Node.js 版本 6+。
- Vue-server-renderer 和 Vue 必须匹配版本。

6.6.9 渲染一个 Vue 实例

```
// 第 1 步：创建一个 Vue 实例
const Vue = require('Vue')
const app = new Vue({
  template: `<div>Hello World</div>`
})
// 第 2 步：创建一个 renderer
const renderer = require('Vue-server-renderer').createRenderer()
// 第 3 步：将 Vue 实例渲染为 HTML
renderer.renderToString(app, (err, html) => {
  if (err) throw err
  console.log(html)
  // => <div data-server-rendered="true">Hello World</div>
})
```

与服务器集成。

在 Node.js 服务器中使用时相当简单直接，例如 Express：

```
npm install express --save
const Vue = require('Vue')
const server = require('express')()
const renderer = require('Vue-server-renderer').createRenderer()
server.get('*', (req, res) => {
  const app = new Vue({
    data: {
      url: req.url
```

```
    },
    template: `<div>访问的 URL 是： {{ url }}</div>`
  })
  renderer.renderToString(app, (err, html) => {
    if (err) {
      res.status(500).end('Internal Server Error')
      return
    }
    res.end(`
      <!DOCTYPE html>
      <html lang="en">
        <head><title>Hello</title></head>
        <body>${html}</body>
      </html>
    `)
  })
})
server.listen(8080)
```

使用一个页面模板。

为了简单些，你可以直接在创建 renderer 时提供一个页面模板。多数时候，我们会将页面模板放在特有的文件中，例如 index.template.html，实际上这个就是前后端通用的一个模板：

```
<!DOCTYPE html>
<html lang="en">
  <head><title>Hello</title></head>
  <body>
    <!--Vue-ssr-outlet-->
  </body>
</html>
```

注意<!--Vue-ssr-outlet-->的注释：这里将是应用程序 HTML 标记注入的地方。

然后，我们可以读取和传输文件到 Vue renderer 中：

```
const renderer = createRenderer({
  template:       require('fs').readFileSync('./index.template.html',
'utf-8')
})
renderer.renderToString(app, (err, html) => {
  console.log(html) // will be the full page with app content injected.
})
```

6.6.10 一个例子

假设在 a.Vue 里面通过 Ajax 取来数据，然后做展示，用 Vuex 来看看怎么搞。如果不用 Vuex，我们取完数据会放到 Data 里面，然后拿到数据做 v-for 渲染如下：

```
...
mounted() {
    Vue.axios.get('http://localhost:5000/data').then((response) => {
        const list = response.data.data.liveWodList
        this.newList = list
    })
}
...
```

如果用 Vuex 呢？显然它不在你的 a.Vue 里面，所以你应该告诉他，来数据了，快收一下，怎么通知呢，这就涉及到 Vuex 的第一个操作：commit。这里的这个操作，对应 Vuex 的核心概念之一：Mutations（变化）！它的作用就是通知 Vuex 要搞事情了，比如删除数据、增加数据等，代码是这样的 this.$store.commit('setData', list)，这里有两个参数，第一个参数是要搞的事情，第二个参数是具体的数据。数据存哪了？你的数据是来了，我要有地方来接收数据吧，接收数据的地方对应 Vuex 的核心概念之二 State（状态），就是所有需要变化的东西都存在我这里。代码是这样的：

```
function setData(state, data) {
    state.list = data
}
```

怎么拿到数据？有放肯定有取，数据存在 State，取也是从这里取。取数据就对应 Vuex 的核心概念之三 Getters，代码是这样的：

```
const getters = {
    list: state => state.list
}
```

目录结构如下：

```
store
|____modules
| |____list.js
|____mutation-types.js
```

```
|____store_index.js
```

这是在原目录的基础上增加的目录. 这个目录结构有什么好处？ 这是 Vuex 在真正项目中用到的，分模块，每个模块一个文件（modules），首先我们看下面这个文件：store/mutation-types.js。这个文件的结构比较简单，代码如下：

```
export const LIST = {
    GET_DATA: 'getData',
    ADD_DATA: 'addData'
}
```

功能是定义常量。常量的作用不用细说，防止手写写错。实际开发中，应该是一个模块、一个常量，现在只有一个 LIST，未来可能会多增加 NEWS/USER 等，也是一个模块，一个常量对象。再看 modules/list.js，代码如下：

```
import {
    LIST
} from '../mutation-types'

const state = {
    list: []
}

const mutations = {
    [LIST.GET_DATA](state, data) {
        state.list = data
    }
}
const getters = {
    list: state => state.list
}
```

这里面就有对应的三个概念 state/mutations/getters，可以和我上面说的对比一下。现在看代码，应该很清晰了。store/store-index.js 为入口文件，里面主要是引入各配置，供 Vue 使用。

注意：这个文件的引入是在 src/index.js 的里面。

三个概念的对应关系如图 6-2 所示。

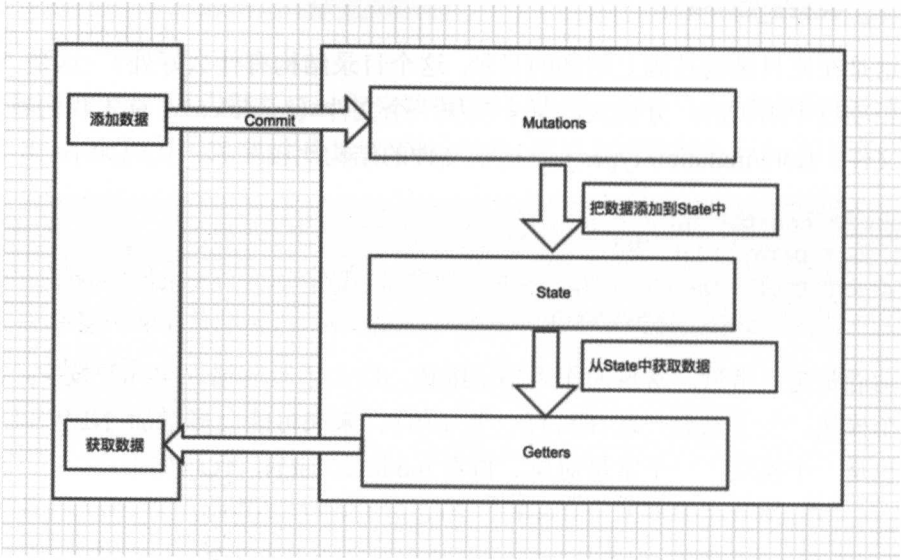


图 6-2

6.7 Vue 的 pre-render

pre-render、ssr、pwa 这类优化的技术手段越来越受到人们的关注，本章主要介绍在 Vue 中如何使用 pre-render 技术。

在介绍 pre-render 之前，我们先要弄清楚这些技术解决了什么问题。主要有两大问题，第一个问题是 seo，这也是很多人对 spa（单页网站应用）的担忧，人们希望自己的网站被搜索引擎收录，但是一些搜索引擎对 spa 支持的不好。pre-render、ssr 能够解决这方面的问题。另一个问题是 slow client，也就是网页打开的速度。众所周知，网页会先解析 HTML，所以如果 HTML 是后台直出的话，用户会更快的看到页面。

伴随着 spa 应用的崛起，ssr 一直视为 spa 的一大痛点，个人认为应该理性看待 ssr，ssr 所解决的 seo 问题，可以通过 pre-render 解决，而 slow client 问题，可以通过 pwa 来改善，ssr 带来代码复杂度上的成本是巨大的，而单独的进行 ssr 又未必能达到理想的效果。

pre-render 又分两种，一种是纯 pre-render，这里面以 prerender.io 为代表，主要是用来解决 seo 的问题，原理就是当请求打到服务端时判断是否是网络爬虫的请求，如果是网络爬虫的请求，我们会重定向到由 phantomjs 渲染页面的服务中。

由 phantomjs 返回完整网站的 HTML。

还有一种是混合 pre-render，它和纯 pre-render 的区别是纯 pre-render 是运行在服务端的，而混合 pre-render 是在 spa 构建 HTML 的时候将页面直出，所以有一部分页面并不适用于混合 pre-render，比如：动态页面（如用户个人中心，每个用户看到的都不一致）。混合 pre-render，我们主要推荐的是 prerender-spa-plugin，也是官网推荐的 Vue pre-render 解决方案。

prerender-spa-plugin 是 webpack 的构建插件，需要结合 webpack 才能进行构建。

按照官网的指示，我们可以 clone prerender-spa-plugin 项目来一探究竟。

npm build 之后我们可以看到每个路由，prerender-spa-plugin 都为我们生成了对应的 HTML，如图 6-3 所示。

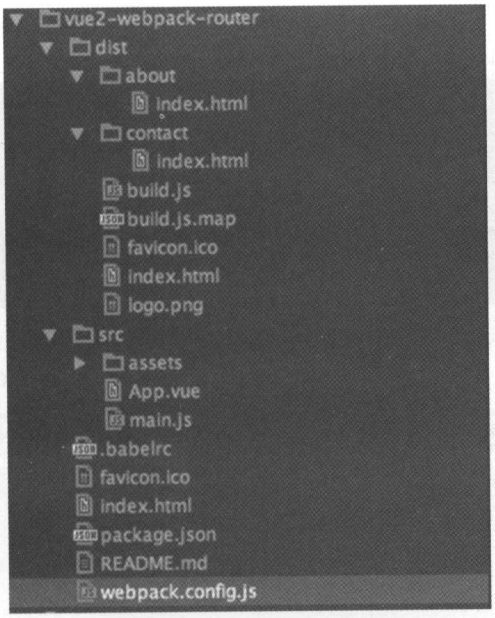


图 6-3

读 者 圈

第 7 章　原理解析

7.1　Virtual DOM 原理

React 以 Virtual DOM 享誉世界，或者 Virtual DOM 因为 React 被人揪出来反复研讨。看上去这个机制非常重要，React 官方网站最开始都以 Virtual DOM 为主要的亮点之一。继而活跃于各种 mvvm 的框架中，造福前端众人。其相互关联如图 7-1 所示。

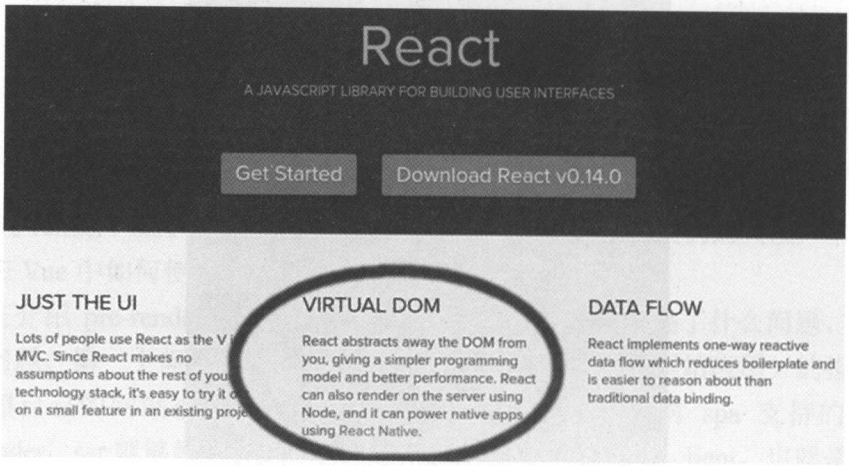


图 7-1

那么，Virtual DOM 是什么？下面我们从 DOM 开始看。

7.1.1　DOM

DOM 全称是 Document Object Model，是一种结构化模型，对于 web 开发者来说，结构化的文本是 html 代码，结构模型就是 HTML DOM。HTML 的元素就是 DOM 结构的节点（nodes）。

以浏览器为例，html 代码只是一堆字符串，但是浏览器进行 UI 解析后，变

成了内存中的一颗 DOM 树，进而展示为我们看到的页面。我们开发中所操作的 DOM，就是这颗内存中的树。记不记得 DOM 树的重绘和回流,可以理解为浏览器的类 Virtual DOM 装置所做的工作之一。

HTML DOM 的操作靠的是 JS API，如 getElementById；跟 DOM 关系最大的就是 JS 了。

不管怎样，我们如果要改变页面内容，就要操作 DOM：

```
var item = document.getElementById("myLI");
item.parentNode.removeChild(item);
```

像上面这样，就可以改变页面展示了。时至今日，浏览器计算能力越来越强，我们的页面元素也越来越丰富，DOM 操作无可避免也越来越多，尤其是出现了以交互体验至上的 SPA（单页面应用）后，使得 DOM 操作更是成倍增长，趋势如此，所以开发起来会很困难。想象一下，一个由上万 div 组成的 DOM，要给它们绑定事件，有代理的，有自己的，更新内容后还得加上新的，如果都用 DOM 操作，开发维护起来将会是无穷地狱。

两个突出的问题：

- 难以控制。如果要调整一个事件响应方法，要深入到代码中，找到关联的函数，分析依赖的部分，小心的调整，耗时且出 bug 可能性非常的高；
- 效率低下。写的时候效率低，改的时候效率低，其他人接手时，依然效率很低。

这一次来救我们的是 Virtual DOM，一起来看看它如何运作，看一个例子：

```
var CommentBox = React.createClass({
  getInitialState: function() {
return {secondsElapsed: 0};
},
tick: function() {
this.setState({secondsElapsed: this.state.secondsElapsed + 1});
},
componentDidMount: function() {
this.interval = setInterval(this.tick, 1000);
},
componentWillUnmount: function() {
clearInterval(this.interval);
},
  render: function() {
    return (
```

```
        <div className="commentBox">
          Hello, world! I am a CommentBox. {this.state.secondsElapsed}s
passed!
        </div>
      );
    }
  });
```

每隔一秒，更新 commentBox 节点。显而易见的逻辑，每一秒会执行 tick 函数，改变 secondsElapsed 的值，进而导致节点内容更新；这是 React 的写法，经过前面几章的介绍，相信应该很好理解，那么这个过程中 Virtual DOM 是怎么生效的呢？

首先 React 持有现状 React DOM Tree，当属性值变化时，React 会收集到变化来构造新的 Reac DOM Tree，在新的 React DOM Tree 中属性值变化会造成这个节点被标注；新的 Tree 更新完之后会跟现状 Tree 做对比，经过一系列的 diff 算法处理及更新，该有的更新就渲染出了。

在此过程 React 构造的树可称之为 React DOM Tree，它的基本节点是由 React 组件构成的，每个节点都包括一系列的属性值（如 key，type），子组件等；任意一个组件，如果调用它的 setState，那这个节点会被标记为脏节点，在事件处理结束后，React 会找到所有的脏节点并且重新渲染。

过程是不是很简单，Virtual DOM 的引入给我们带来了处理复杂渲染的机制：

- 我们无需关注 DOM 区别，只需要 focus 在数据的更改上；
- 我们无需直接操作 DOM，使得代码层面界面更新变得更简单；也可以说是 React 与 Virtual DOM 双赢的结果；
- 极大提高了界面开发的效率。

如果要细说 React DOM Tree 和 diff 方法，那就会相对复杂一些，对了，再说几点跟 Virtual DOM 相关的调优原则：

- 组件实现 shouldComponentUpdate 方法，主动把控什么时候值得更新；
- 避免跨级别调整 DOM 节点，保持稳定的 DOM 结构会有助于性能的提升；
- 同级几点之间减少数量过大或者过于频繁的位置切换操作。

关于框架，针对 Virtual DOM 做的调优，目的是加快渲染过程，有兴趣的可以查一下 React 或 Vue 源代码，实践中体验一下。

7.1.2　Virtual DOM 算法

DOM 的操作与计算，本身是很慢的，任何一个 DOM 的标准属性都有一大堆，这也是 HTML 标准中规定的，再加上读取更新 DOM 的耗时，如果要做完整的对比和操作，那将是一个灾难的开始。然而数据层面的比较要简单得多，而且基于一些约定，我们可以做到只比对少数关键数据属性即可；再加上不考虑跨层的比对，时间复杂度会到零。

如图 7-2 所示的是一张深度优先遍历的过程，在此过程中，我们的程序需要记录的是节点的变化。

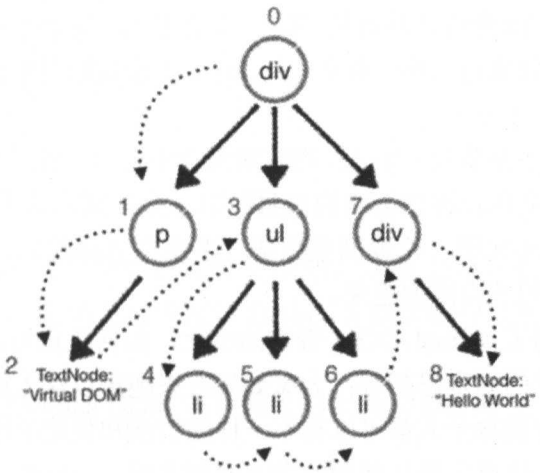


图 7-2

节点变化一般分为三种情况：

- 节点被删除；
- 节点修改了（如：节点的宽度属性变化，位置变化等）；
- 节点新增了子节点。

记录变化之后，还存在一些问题，比如，如果节点只是顺序变化，记录后挨个挪动效率比较低，简单处理可能认为这些节点的父节点需要完全重新渲染。所以框架里很多会对此类变化做优化，如图 7-3 所示。

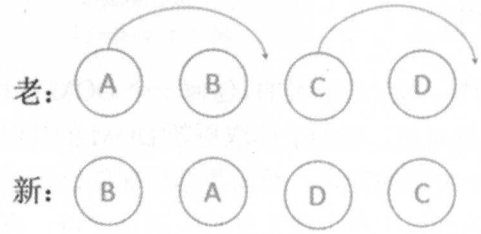


图 7-3

不做优化的处理方式，对比 A 和 B 发现不一致，那么需要更新第一个子节点为 B，删除原节点 A，以此类推，需要把所有子节点删除，重新渲染更新；这样在大批量的计算中，浪费也是显而易见的，改造方式介绍如下：

- 给同级的子元素打上唯一标识 id，根据 id 来判断老集合与新集合之间节点的差异；
- 老集合中取一个节点，如 A；判断新集合中是否存在；如存在，再判断位置是否需要变化；再做相应的调整；
- 老集合中循环完毕，位置调整完毕，多余节点删除后，再循环新集合，把需要增加的节点添加进来。

以上，大致介绍了 Virtual DOM 带来的变化，如何优化自己的 DOM 操作，及 Virtual DOM 算法的基本思想等；大家如果有兴趣可以深入研究一下树的对比技巧和算法。在这方面每个人都可以有不一样的看法和优化手段。掌握原理能让我们的代码更高效，也更容易从问题的表面看到本质，当然能力也就随之提高，这也是我们认可的工程师成长的重要方式。

7.2 Vue 精髓之响应式数据流

7.2.1 数据流演进史

前端数据流的概念可以从 2010 年诞生的 backbone 典型的 MVC 框架说起，随着前端的业务越来越复杂，前端的技术也在不断的演进中。在 backbone 提出的前端 mvc 概念后，又诞生了 knockout，angular 这类 MVVM 模式的框架，再之后就是目前流行的基于 Virtual DOM 的 Flux。每个框架管理数据流的方式都不尽相同，接下来我们引用 Vue 作者尤雨溪的一句话，来开启今天 Vue 的响应式数据流探索之旅。

Vue 2.0 的实现有与众不同的地方。和 Vue 的响应式系统结合在一起之后，它可以让你不必做任何事就获得完全优化的重渲染。由于每个组件都会在渲染时追踪其响应依赖，所以系统精确地知道应该何时重渲染，应该重渲染哪些组件。不需要 shouldComponentUpdate，也不需要 immutable 数据- it just works。

为了更好地理解响应式数据流，我们首先介绍 Vue。

7.2.2　Vue 和 React 介绍

目前前端社区比较推崇的框架有 Vue 和 React，公司内部许多端都自发地将原有的老技术方案（widget + jQuery）迁移到 Vue 和 React 上了。我觉得 Vue 和 React 有以下几点优势：

- 首先它们都有完整的组件化方案；
- Virtual DOM（前端性能提升利器）；
- 成熟的社区生态。

7.2.3　Vue 的响应式数据流的优势

Vue 和 React 都是前端的组件化框架，功能上大同小异，本质上就是借助 Virtual DOM 帮助开发者管理混乱的 DOM，并提供给开发者像操作状态机一样操作页面的能力。

但是，Vue 的 Virtual DOM 不是简单的 Virtual DOM，它结合了响应式数据流的能力。

我们看一下第三方的性能分析，如图 7-4 所示。

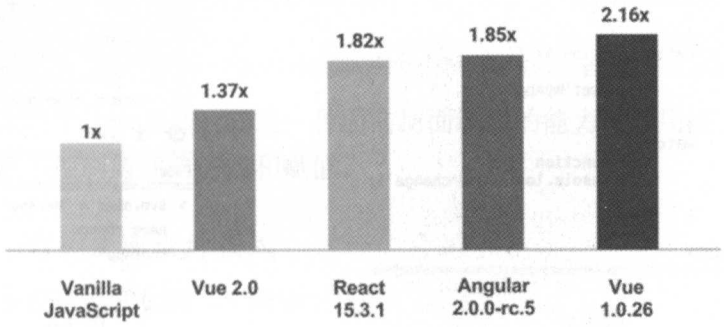


图 7-4

除了性能，最大的优势是减轻了开发者的负担，开发者大多数情况下不需要依赖 shouldComponentUpdate，也不需要依赖 immutable 数据去判断组件是否需要重新渲染，Vue 会帮你做好这件事。

举个例子来说明这两个的 Virtual DOM 的不同之处：

开发者就像一个老师，Vue 和 React 这两个学生要做的事就是根据老师给出的长宽画出对应的长方形。每当老师改变给出的长和宽时，Vue 能够自己发现长和宽变没变，需不需要重新画；React 则需要老师告诉它长和宽变了，需要重新画了。

在详细介绍实现细节之前，我们先介绍一些预备知识来帮助大家更好的理解细节。

介绍一个 Vue 例子，如图 7-5 和图 7-6 的示。

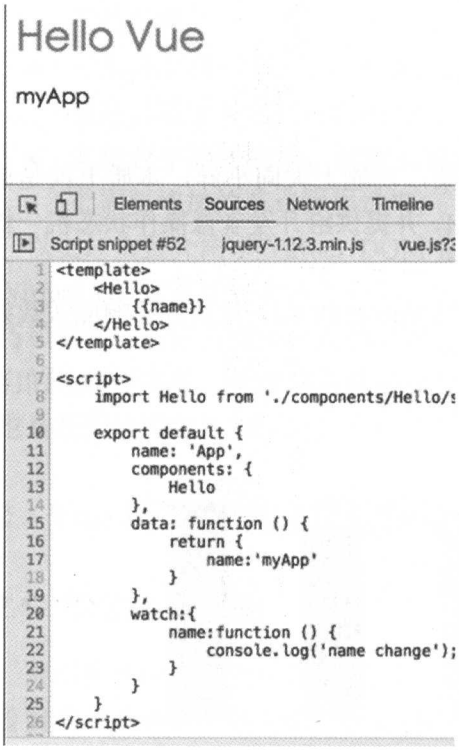


图 7-5

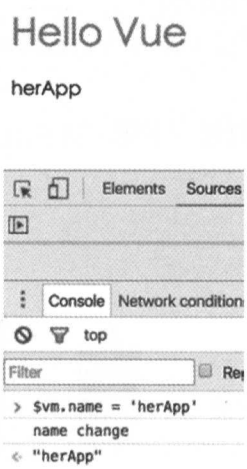


图 7-6

上面的例子我们初始化了一个 Vue 组件，当我们改变这个组件的状态时，页面的内容也会随之改变，这中间并不需要我们手动的去操作页面上的 DOM 元素。

同时，我们注意到 Vue 提供了一个语法糖——watch，这个就是我们今天要讲的 Vue 响应式数据流的主角！代码很简单，就是组件的状态 name 改变的时候输出的一句话“name change”。下面我们会向大家解释清楚为什么这个 watch 这么重要，以及它和 Vue 的响应式数据流有什么关系。

7.2.4　Object.defineProperty 与订阅发布设计模式

7.2.4.1　Object.defineProperty

JavaScript 提供一个非常强大的方法 Object.defineProperty，它可以定义当某一个值访问和赋值时会先执行自定义的钩子方法。

一个简单的 Object.defineProperty 例子如下：

```
var obj = {};
var initValue = 'hello';
Object.defineProperty(obj,"newKey",{
    get:function (){
        //当获取值的时候触发的函数
        return initValue;
    },
    set:function (value){
        //当设置值的时候触发的函数,设置的新值通过参数 value 拿到
        console.log(value)
        initValue = value;
    }
});
//获取值
console.log( obj.newKey );  //hello
//设置值
obj.newKey = 'change value'; //change value
```

这个方法给予 JavaScript 开发一种面向切面编程的能力，使用该方法我们能够隐式、自然的控制属性的访问和赋值。

7.2.4.2　订阅发布设计模式

订阅发布设计模式如图 7-7 所示。

订阅发布是一个非常常见的设计模式，原理也非常简单，就是订阅者订阅信息，然后发布者发布信息通知订阅者更新。

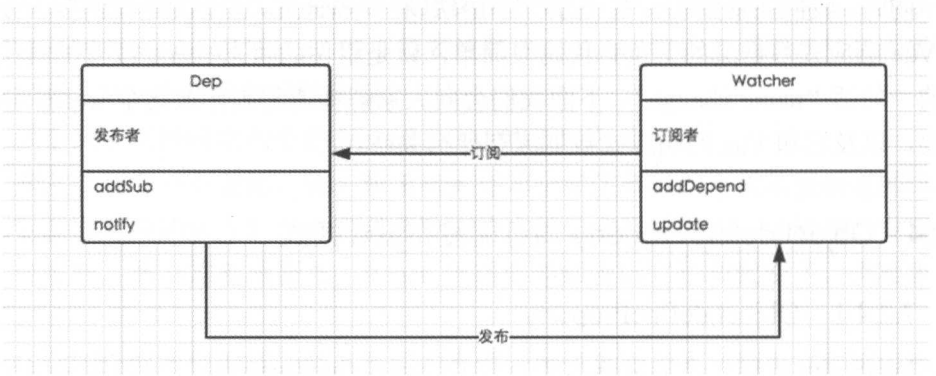


图 7-7

7.2.5 Vue 源码

前面铺垫这么多就是希望大家能理解接下来要讲的响应式数据流。

7.2.5.1 Vue 初始化流程

Vue 初始化流程图如图 7-8 所示。

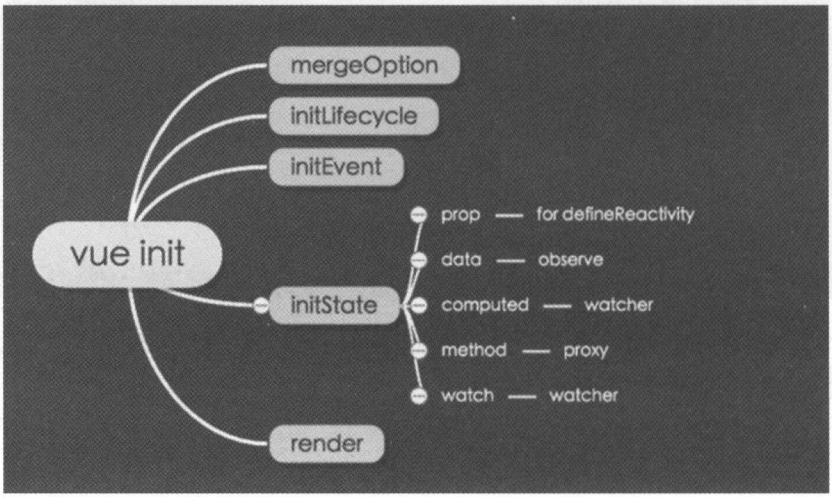


图 7-8

Vue 的初始化会执行一系列的方法，这里我们主要介绍 Vue 的 initState 方法。prop 和 data 都是组件的属性，prop 通常是由父组件传递下来的，data 是组件自身

定义的，Vue 不推荐你去改组件传递下来的 prop，因为那样会带来不必要的复杂度。

7.2.5.2　整个数据流的流程图

我们先看一下整个数据流的流程图（如图 7-9 所示），对相关节点有个基本的概念。

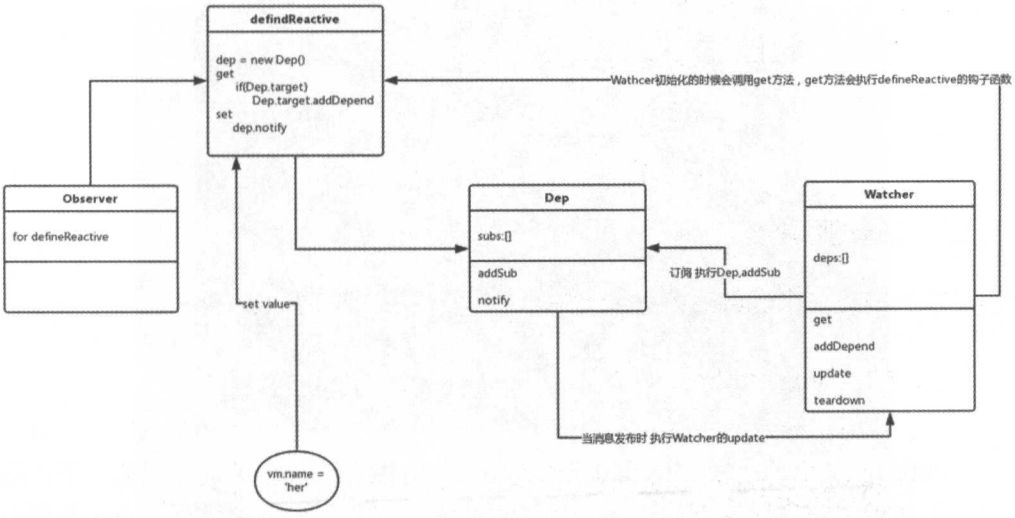


图 7-9

7.2.5.3　Observer

上面提到的 Prop 和 data 的最终归宿都是递归执行 defineReactive 的方法：

```
function initProps (vm: Component) {
  const props = vm.$options.props
  if (props) {
    const propsData = vm.$options.propsData || {}
    const keys = vm.$options._propKeys = Object.keys(props)
    /*...*/
    for (let i = 0; i < keys.length; i++) {
      /*...*/
      // 遍历所有属性并执行defineReactive
      defineReactive(vm, key, validateProp(key, props, propsData, vm))
      /*}*/
    }
    /*observerState.shouldConvert = true*/
  }
}
```

那 defineReactive 方法做了什么呢？

```
export function defineReactive (
  obj: Object,
  key: string,
  val: any,
  customSetter?: Function
) {
  const dep = new Dep()
/*...*/

  let childOb = observe(val)
    // 将每个属性都用Object.defineProperty包装一下,
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter () {
      const value = getter ? getter.call(obj) : va
        // 能够知道是谁访问了这个属性
      if (Dep.target) {
        dep.depend()
        if (childOb) {...}
        if (Array.isArray(value)) {...}
      }
      return value
    },
    set: function reactiveSetter (newVal) {...}
  })
}
```

defineReactive 会用 Object.defineProperty 将组件的每个属性都包装一下，这样，谁访问了这些属性，谁重新赋值了这些属性，我们都能追踪到。Vue 里面有一个 Observe 类，所有的 prop 子属性和 data 本身都会带有一个 Observer 对象。Observer 的构造函数如下：

```
constructor (value: any) {
  this.value = value
  this.dep = new Dep()
  this.vmCount = 0
    // value 是 Vue的属性, this是Observer实例
  def(value, '__ob__', this)
  if (Array.isArray(value)) {
    const augment = hasProto
      ? protoAugment
      : copyAugment
    augment(value, arrayMethods, arrayKeys)
    this.observeArray(value)
  } else {
    this.walk(value)
  }
}
```

在控制台我们可以看到每个属性下都有__ob__，这说明这个属性已经被包装成 Observer 对象了，所有的访问和赋值都能追踪到，这里面也保存着所有订阅该 Observer 的订阅者 Watcher。

7.2.5.4　Watcher

我们看以下 Watcher 的构造函数：

```
/** A watcher parses an expression, collects dependencies
export default class Watcher {
  vm: Component;
  expression: string;
  cb: Function;
  // 构造函数
  constructor (
    vm: Component,
    // 构造函数中的关键参数 expOrFn,可以是函数也可以是字符串
    expOrFn: string | Function,
    cb: Function,
    options?: Object = {}
  ) {
    this.vm = vm
    vm._watchers.push(this)
    // 如果expOrFn是函数就执行该函数,如果是字符串则解析字符串
    if (typeof expOrFn === 'function') {
      this.getter = expOrFn
    } else {
      this.getter = parsePath(expOrFn)
    }
    // 初始化的时候用this.get()取一次值,建立订阅发布关系
    this.value = this.lazy
      ? undefined
      : this.get()
  }
```

Watcher 支持 watch 一个表达式或者是一个方法。Watcher 在构造的时候会先获取一次 expOrFn 的值，下面我们把 expOrFn 称为 Watcher 的 Getter。

7.2.5.5　Dep

还有一个关键的类是 Dep，这个类会帮助我们的属性记录下所有的 Watcher，每个属性都有自己的 Dep 实例，同时 Vue 的 Watcher 访问属性的时候，Dep 会作为一个全局变量将自身的 target 属性指向访问的 Wathcer。会执行下面的方法：

```
Dep.target = null
const targetStack = []
export function pushTarget ( target: Watcher) {
    // 取值执行this.get()前,先将Dep类作为全局对象,将Dep.target
    // 这个静态属性执行 Watcher自身
  if (Dep.target) targetStack.push(Dep.target)
  Dep.target = _target
}
export function popTarget () {
  Dep.target = targetStack.pop()
}
```

同时我们再回来看 defineReactive 这个重要的方法：

```
export function defineReactive (
  obj: Object,
  key: string,
  val: any,
  customSetter?: Function
) {
  const dep = new Dep()
/*...*/

  let childOb = observe(val)
    // 将每个属性都用Object.defineProperty包装一下,
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter () {
      const value = getter ? getter.call(obj) : va
        // 能够知道是谁访问了这个属性
      if (Dep.target) {
        dep.depend()
        if (childOb) {...}
        if (Array.isArray(value)) {...}
      }
      return value
    },
    set: function reactiveSetter (newVal) {...}
  })
}
```

当 Watcher 访问组件的属性时，通过 Dep.target，Vue 可以知道是 Watcher 访问的，这样当 Vue 自己的 Watcher 访问属性的时候会被记录成订阅者，而我们访问的时候 Vue 不会执行多余的代码。这是一个很精妙的设计，将 Object.defineProperty 与订阅发布设计模式结合起来了。

7.2.6 Vue 的 render 函数就是 Watcher 的 expOrFn

理解了以上 Vue 是如何将 Object.defineProperty 与订阅发布设计模式结合起来的，然后我们再举一反三：Vue 的 render 函数如果就是 Watcher 的 expOrFn 会怎

么样？回到 Vue 的源码里：

```
callHook(vm, 'beforeMount')
vm._watcher = new Watcher(vm, () => {
    // 将render函数用_update和_render包装一下，
    // 作为 watcher的expOrFn
  vm._update(vm._render(), hydrating)
}, noop)
hydrating = false
// manually mounted instance, call mounted on s
// mounted is called for render-created child c
if (vm.$vnode == null) {
  vm._isMounted = true
  callHook(vm, 'mounted')
}
```

这里的 vm._render 就是 render 函数的一个封装，我们可以看到：本质上 Vue 的 render 函数就是 Watcher 的 expOrFn。那初始化的时候我们会先执行一边 render 函数，在执行 render 函数的过程中访问了哪些组件的属性，Vue 都会用上面提到的方法帮我们把依赖记录下来。所以当这个属性变化的时候，自然而然，就像文章开头的 watch 一样，我们会重新 render 一次（开头的例子是输出“name change”）。

讲到这里大家应该都能够明白 Vue 的响应式数据流是如何实现的。同时我们能够发现 Vue 提供给我们的许多语法糖都是同样的道理，比如 Vue 的 computer 就是将 computer 函数作为 Watcher 的 expOrFn。希望大家在理解 Vue 响应式数据流的基础上能够更加自信、灵活和稳健的使用这个优秀的框架。

7.3　Vuex2.0 源码解析

7.3.1　Vuex 的含义

我们知道 Vue 是渐进式的 js 框架，其核心关注在数据管理上，通过构建数据驱动 web 界面，那么对于数据和状态的规范管理就变得尤为关键和重要。Vuex 是一个专门为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。我们不妨假设一个场景帮助理解 Vuex 的含义。

试想这样一个场景：你是一个小型公司的管理者，为了保证收益所有的资源流动都需要你的监控，从批发材料、生成制作到投入市场运营，每个环节都需要你做好信息资源的融合交换，试想如果未来公司越来越大，更多更具体的部门产

生，而且互相之间的关系也会变得更加错综复杂，这个时候作为管理者的你已经分身乏术，这个时候要怎么办？要么增加管理者，不过这个好像不能本质上解决问题；要么创建一种智能制度去管理。而 Vuex 的产生就是这样一个智能的管理制度，管理 Vue 组件间的资源流动，从而使管理更为轻松和有效，像是一个超级管理者。

7.3.2 源码分析

7.3.2.1 目录结构

目录结构如下：

整个目录结构十分的简洁，主要有四个 js 文件，plugins 目录是 Vuex 的两个内置插件。Vuex2.0 整个源码加起来不过 500～600 行，是个非常轻巧的库。虽然轻巧，但是功能完备，接下来详细介绍其中的实现细节。

7.3.2.2 初始化

Vuex 是基于 ES6 的语法编写的，因此如果你还不太熟悉这部分的语法，建议先了解一下 ES6 的语法，从而帮助你更好地理解这部分的内容。

Vuex 本身是一个库，Vuex 需要接入 Vue 的环境里才能使用，因此，Vuex 设计的开始需要暴露几个 API 给使用者，需要暴露什么 API 呢？暴露几个呢？既然是状态管理，那么需要一个主要的 API 来管理状态，暂时叫它 Store；库需要接入到 Vue 环境里，需要生产一个 install；可能还需要一些语法糖，这部分会在后面详细介绍。

```
export {
  Store,
  install,
  mapState,
  mapMutations,
  mapGetters,
  mapActions
}
```

由上可以一目了然地看到 Vuex 对外暴露的 API，现在详细介绍一下 install 的这个 API。

Vuex 创建了一套安装的“套路”：安装的核心内容是对 Vue 实例接入$store 属性，而在安装之前需要进行环境判定，避免重复安装。

Vuex 的安装源码如下：

```
export function install (_Vue) {
// 如果 Vue 实例存在，不需要再次安装，避免重复安装
  if (Vue) {
    console.error(
      '[Vuex] already installed. Vue.use(Vuex) should be called only
once.'
    )
    return
  }
  Vue = _Vue
  applyMixin(Vue)
}

// 默认安装
if (typeof window !== 'undefined' && window.Vue) {
  install(window.Vue)
}
```

applyMixin 方法是对 $store 属性的注入，注入前同样需要对 Vue 的版本进行区分，从而根据不同版本在不同的生命阶段注入 store 属性。

```
export default function (Vue) {
  const version = Number(Vue.version.split('.')[0]);

  if (version >= 2) {
  // Vue2.0 环境下，在 beforeCreate 阶段进行注入
    const usesInit = Vue.config._lifecycleHooks.indexOf('init') > -1
```

```
    Vue.mixin(usesInit ? { init: VuexInit } : { beforeCreate: VuexInit })
  } else {
    // Vue1.0 环境下对 init 进行重写
    const _init = Vue.prototype._init
    Vue.prototype._init = function (options = {}) {
      options.init = options.init
        ? [VuexInit].concat(options.init)
        : VuexInit
      _init.call(this, options)
    }
  }

  function VuexInit () {
    const options = this.$options
    // store 注入
    if (options.store) {
      this.$store = options.store
    } else if (options.parent && options.parent.$store) {
      this.$store = options.parent.$store
    }
  }
}
```

完成注入工作以后，安装也就告一段落了，接下来介绍使用 Vuex 背后都做了什么工作，也就是 Vuex 的核心内容——store。

7.3.2.3 认识核心——store

在认识核心之前，介绍一下使用 Vuex 的正确姿势：实例化 Store 类，实例化过程中传入一个参数对象，参数对象包括我们定义好的 actions、getters、mutations、state 等，甚至当我们有多个子模块的时候，我们可以添加一个 modules 对象。

看到这里你可能会问上面的这些属性是什么，它们分别是用来做什么的？

store 是 Vuex 设计的一个管理状态的库，一切的核心都应该围绕着状态，那么这个库就需要定义一些类。

- 首先定义一个状态类——state，用来存储各个组件的状态，最终形成一个层级状态树，根据层级关系获取组件的状态；
- 有存储就应该有获取状态的动作，state 树本身可以直接获取，但是取的都是本身存入的值。如果需求上需要进行二次修改，例如进行一次过滤或者加减乘除的计算等，这个时候需要定义一个 getter 类——对一些通用

数据状态的处理，并且这个处理方法是全局使用的；

- 同时还有修改状态问题，Vuex 管理的都是响应式数据状态，直接关系到页面的重新渲染，因此修改需要同步执行，这样一个修改类命名为 mutation；
- 我们还需要定义一个 actions 类——实现异步修改状态的问题，actions 类的实质也是提交 mutation，从而保证准确追踪状态变化；
- 面对庞大的应用，还需要引入 modules 的理念，对嵌套的模块进行 store 分割，每个模块都是一个小型的 store，拥有自己的 state、mutation、action、getters，甚至是再嵌套子模块。

以上的都是 store 的核心概念，也构成了 store 的主体部分。接下来让我们回到 index.js 里的 store，重点看以下部分的源码。首先看下面的构造函数：

```
  constructor (options = {}) {
  // 判定是否按照 Vuex，同时判定环境是否支持 Promise，后续会使用 Promise
    assert(Vue, `must call Vue.use(Vuex) before creating a store
instance.`)
    assert(typeof Promise !== 'undefined', `Vuex requires a Promise
polyfill in this browser.`)

    const {
      state = {},
      plugins = [],
      strict = false
    } = options

    // store 内部状态
    this._committing = false
    this._actions = Object.create(null)
    this._mutations = Object.create(null)
    this._wrappedGetters = Object.create(null)
    this._modules = new ModuleCollection(options)
    this._modulesNamespaceMap = Object.create(null)
    this._subscribers = []
    this._watcherVM = new Vue()

    // 对 store 绑定 dispatch 和 commit 方法
    const store = this
    const { dispatch, commit } = this
    this.dispatch = function boundDispatch (type, payload) {
      return dispatch.call(store, type, payload)
```

```
    }
    this.commit = function boundCommit (type, payload, options) {
      return commit.call(store, type, payload, options)
    }

    // 严格模式
    this.strict = strict

    // 初始化根模块
    // this also recursively registers all sub-modules
    // and collects all module getters inside this._wrappedGetters
    installModule(this, state, [], this._modules.root)

    // 初始化 store VM, which is responsible for the reactivity
    // (also registers _wrappedGetters as computed properties)
    resetStoreVM(this, state)

    // 插件应用
    plugins.concat(devtoolPlugin).forEach(plugin => plugin(this))
  }
```

构造函数一开始需要进行 Vue 环境的二次判定和 ES6 环境的判定，从而确保下面的实例化工作正常进行。

接下来通过 ES6 语法的结构赋值拿到 options 里对应的 state，plugins 和 strict，同时有一些内部属性的声明初始化工作。先对这些内部属性进行初步认识。

- this._committing：标志一个提交状态，其作用是保证对 Vuex 中 state 的修改只能在 mutation 的回调函数中，而不能在外部随意修改 state；
- this._actions：用来存储用户定义的所有 actions；
- this._mutations：用来存储用户定义的所有的 mutations；
- this._wrappedGetters：用来存储用户定义的所有的 getters；
- this._modules：用来存储用户定义的所有 modules；
- this._modulesNamespaceMap：用户存储所有的 modules 对应 namespace 的对象；
- this._subscribers：用来存储所有对 mutation 变化的订阅者；
- this._watcherVM：是一个 Vue 对象的实例，主要是利用 Vue 实例方法 $watch 来观测变化的。

下面介绍两个很重要的方法——commit 和 dispatch，分别看以下各自的源码。

（1）commit。

```
commit (_type, _payload, _options) {
    // 解构定义参数
    const {
      type,
      payload,
      options
    } = unifyObjectStyle(_type, _payload, _options)

    const mutation = { type, payload }
    const entry = this._mutations[type]
    //判定是否 mutation 是否存在
    if (!entry) {
      console.error(`[Vuex] unknown mutation type: ${type}`)
      return
    }
    this._withCommit(() => {
      entry.forEach(function commitIterator (handler) {
        handler(payload)
      })
    })
    // 订阅监听
    this._subscribers.forEach(sub => sub(mutation, this.state))

    if (options && options.silent) {
      console.warn(
        `[Vuex] mutation type: ${type}. Silent option has been removed.
` +
        'Use the filter functionality in the Vue-devtools'
      )
    }
  }
```

前面介绍到 mutation 是对 state 进行状态的修改，而 commit 方法则是 mutation 的内核，所有的修改工作都要提交到这个方法里执行。源码里一开始是一些数据的解析，拿到对应模块的 mutation 后进行一次非空检测，接下来有一个方法——_withCommit，源码如下：

```
_withCommit (fn) {
    const committing = this._committing
    this._committing = true
    fn()
```

```
    this._committing = committing
  }
```

从前面定义的属性来看，this._committing 是提交的状态，这个方法存在的意义就是保证在修改 state 的过程中，this._committing 的值始终是 true。当我们检测 state 的变化时，如果 this._committing 的值不为 true，则能检测到这个状态的修改是有问题的。执行 handler 即是执行了 registerMutation 注册的回调函数，也就是用户自定义的修改 state 的函数。

接下来是 Store 实例提供了 subscribe API 接口，它的作用是订阅（注册监听）store 的 mutation，最后是一个静默状态报警判定。看以下 subscribe api 的源码部分：

```
subscribe (fn) {
  const subs = this._subscribers
  if (subs.indexOf(fn) < 0) {
    subs.push(fn)
  }
  return () => {
    const i = subs.indexOf(fn)
    if (i > -1) {
      subs.splice(i, 1)
    }
  }
}
```

订阅方法很简单，接受一个回调函数作为参数，将参数存入_subscribers 属性里，并返回一个函数，当我们调用这个返回的函数时解除订阅。

（2）dispatch。

```
dispatch (_type, _payload) {
  // 解构定义参数
  const {
    type,
    payload
  } = unifyObjectStyle(_type, _payload)
  // 获取 action 并进行非空判定
  const entry = this._actions[type]
  if (!entry) {
    console.error(`[Vuex] unknown action type: ${type}`)
    return
  }
```

```
    return entry.length > 1
      ? Promise.all(entry.map(handler => handler(payload)))
      : entry[0](payload)
  }
```

dispatch 则是 action 的内核，在源码结构上跟 commit 基本一致，差别在执行核心方法上，当 action 数组的长度为 1 时直接调用 entry[0] (payload)，这个就相当于 registerAction 注册的回调函数，即用户自定义的 action 回调函数，反之，通过 promise.all 进行并行执行。

介绍完两个函数之后，接着往下看，this.strict 表示是否开启严格模式，在严格模式下会观测所有的 state 的变化，建议在开发环境时开启严格模式，线上环境要关闭严格模式，否则会有一定的性能开销。

构造函数介绍到这里还只是一些前期的准备工作，下面的内容才是 Store 的核心内容，介绍如下。

（1） installModule。

先来看源码：

```
function installModule (store, rootState, path, module, hot) {
  // 定义根变量，在 store 里获取 namespace
  const isRoot = !path.length
  const namespace = store._modules.getNamespace(path)

  // 模块在 namespaceMap 里进行注册
  if (namespace) {
    store._modulesNamespaceMap[namespace] = module
  }

  // 在状态树里设置当前模块的状态
  if (!isRoot && !hot) {
    const parentState = getNestedState(rootState, path.slice(0, -1))
    const moduleName = path[path.length - 1]
    store._withCommit(() => {
      Vue.set(parentState, moduleName, module.state)
    })
  }
  // module store 本地化
  const local = module.context = makeLocalContext(store, namespace,
path)
  // 注册 mutation
  module.forEachMutation((mutation, key) => {
    const namespacedType = namespace + key
```

```
      registerMutation(store, namespacedType, mutation, local)
    })
    // 注册 action
    module.forEachAction((action, key) => {
      const namespacedType = namespace + key
      registerAction(store, namespacedType, action, local)
    })
    // 注册 getter
    module.forEachGetter((getter, key) => {
      const namespacedType = namespace + key
      registerGetter(store, namespacedType, getter, local)
    })
    // 子模块递归安装
    module.forEachChild((child, key) => {
      installModule(store, rootState, path.concat(key), child, hot)
    })
  }
```

安装模块的第一步是根据当前模块进行 namespaceMap 注册（如果没有注册过），然后将当前模块的状态注入到全局状态树里。全局状态树注入的过程中，需要获取当前模块的父模块状态，同时获取当前模块名称，最后通过_withCommit 进行提交修改。

上述配置完成后对当前模块进行配置本地的 dispatch，commit，getters 和 state，即 makeLocalContext，该方法源码如下：

```
  function makeLocalContext (store, namespace, path) {
    const noNamespace = namespace === ''

    const local = {
      dispatch: noNamespace ? store.dispatch : (_type, _payload, _options)
=> {
        const args = unifyObjectStyle(_type, _payload, _options)
        const { payload, options } = args
        let { type } = args

        if (!options || !options.root) {
          type = namespace + type
          if (!store._actions[type]) {
            console.error(`[Vuex] unknown local action type: ${args.type},
global type: ${type}`)
            return
          }
        }
```

```
        return store.dispatch(type, payload)
      },

      commit: noNamespace ? store.commit : (_type, _payload, _options) =>
{
        const args = unifyObjectStyle(_type, _payload, _options)
        const { payload, options } = args
        let { type } = args

        if (!options || !options.root) {
          type = namespace + type
          if (!store._mutations[type]) {
            console.error(`[Vuex]    unknown    local    mutation    type:
${args.type}, global type: ${type}`)
            return
          }
        }

        store.commit(type, payload, options)
      }
    }

    // getters 和 state 通过 defineProperties 的方式进行定义，因为可能会被修改
    Object.defineProperties(local, {
      getters: {
        get: noNamespace
          ? () => store.getters
          : () => makeLocalGetters(store, namespace)
      },
      state: {
        get: () => getNestedState(store.state, path)
      }
    })

    return local
  }
```

在配置 dispatch 和 commit 的过程中参考 namespace，没有 namespace 就使用根路径上的 dispatch 和 commit，反之则通过传入自身模块的参数配置为本地的方法。而在 getters 和 state 的配置方面，需要通过 Object.defineProperties 的方式进行，因为 getters 和 state 是可能会被修改的属性。

在进行完本地化的配置后，依次对该模块进行 mutation、action 和 getter 的注

册工作。分别看一下各自的源码：

```
function registerMutation (store, type, handler, local) {
  const entry = store._mutations[type] || (store._mutations[type] = [])
  entry.push(function wrappedMutationHandler (payload) {
    handler(local.state, payload)
  })
}
```

mutation 的注册函数很简单，通过当前模块的 type 从 store._mutation 属性上获取对应的 mutation 数组，同时将该模块的 mutation 包装函数添加到这个数组里即可。包装函数的执行就是用户自定义的回调函数。

```
function registerAction (store, type, handler, local) {
  const entry = store._actions[type] || (store._actions[type] = [])
  entry.push(function wrappedActionHandler (payload, cb) {
    let res = handler({
      dispatch: local.dispatch,
      commit: local.commit,
      getters: local.getters,
      state: local.state,
      rootGetters: store.getters,
      rootState: store.state
    }, payload, cb)
    if (!isPromise(res)) {
      res = Promise.resolve(res)
    }
    if (store._devtoolHook) {
      return res.catch(err => {
        store._devtoolHook.emit('Vuex:error', err)
        throw err
      })
    } else {
      return res
    }
  })
}
```

action 的注册跟 mutation 很相似，区别在包装函数内部，这个函数执行的时候会调用 action 的回调函数，传入一个 context 对象，这个对象包括了 store 的 commit 和 dispatch 方法、getter、当前模块的 state 和 rootState，等等。接着对这个函数的返回值做判断，如果不是一个 Promise 对象，则调用 Promise.resolve（res）给 res

包装成一个 Promise 对象。最后判断 store._devtoolHook，这个只有当用到 Vuex devtools 开启的时候，我们才能捕获 promise 的过程。action 的包装函数最后返回 res，它就是一个地地道道的 Promise 对象。

```
function registerGetter (store, type, rawGetter, local) {
  if (store._wrappedGetters[type]) {
    console.error(`[Vuex] duplicate getter key: ${type}`)
    return
  }
  store._wrappedGetters[type] = function wrappedGetter (store) {
    return rawGetter(
      local.state,
      local.getters,
      store.state,
      store.getters
    )
  }
}
```

在注册 getters 的时候，getters 的 key 不能重复，然后将该模块的 local.state，local.getters，store.state 和 store.getters 进行包装，添加到 store._wrappedGetters 对象里完成注册。如果当前模块还有子模块，那么重新递归执行 InstallModule 方法即可。

（2） resetStoreVM。

在 Vuex 中，我们在组件中通过 this.$store.getters.xxx 可以访问到对应 getter 的回调函数，那么我们需要把对应 getter 的包装函数的执行结果绑定到 store 属性上。这部分逻辑就在 resetStoreVM 方法里：

```
function resetStoreVM (store, state, hot) {
  const oldVm = store._vm

  // 绑定 store.getters
  store.getters = {}
  const wrappedGetters = store._wrappedGetters
  const computed = {}
  forEachValue(wrappedGetters, (fn, key) => {
    // computed 暂存每个 getter 包装函数的执行结果
    computed[key] = () => fn(store)
    Object.defineProperty(store.getters, key, {
      get: () => store._vm[key],
      enumerable: true
```

```
    })
  })

  // 打开 silent，实例化一个 Vue 的实例
  const silent = Vue.config.silent
  Vue.config.silent = true
  store._vm = new Vue({
    data: {
      $$state: state
    },
    computed
  })
  Vue.config.silent = silent

  // 开启严格模式
  if (store.strict) {
    enableStrictMode(store)
  }

  // 清空旧的 _vm 的状态，并销毁这个对象
  if (oldVm) {
    if (hot) {
      store._withCommit(() => {
        oldVm._data.$$state = null
      })
    }
    Vue.nextTick(() => oldVm.$destroy())
  }
}
```

首先，暂存 store._vm 对象，接下来是定义初始化 store.getters：遍历 store._wrapperGetters 对象，拿到每个 getter 的包装函数，并且将包装函数暂存在 computed 里，接着用 Object.defindProperty 方法为 store.getters 定义 get 方法，也就说明了当我们调用 this.$store.getters.xxx 这个方法的时候，实际上访问的是 store._vm[xxx]。

其次，暂存 Vue 的配置 silent，并赋值 silent 为 true，进行实例化过程，最后还原 silent 的默认值。打开 silent 的目的是取消这个 _vm 的所有日志和报警。实例化过程中传入 store 的状态树 state 和上面暂存数据的 computed 变量，这样访问 this.$store.getters.xxx 最后的本质是 computed[xxx]，也就是包装函数的执行结果——回调函数。最后是严格模式开启，看以下严格模式的代码：

```
  function enableStrictMode (store) {
    store._vm.$watch(function () { return this._data.$$state }, () => {
      assert(store._committing, `Do not mutate Vuex store state outside
mutation handlers.`)
    }, { deep: true, sync: true })
  }
```

严格模式做的事情很简单，检测 store._vm.state 的变化，如果 state 的变化不是通过 mutation 的回调函数改变的，而是直接在外部修改的，那么 store._committing 为 false，这样就会抛出一条错误。严格约定 state 的修改只能在 mutation 的回调函数里。

最后，是清空旧的_vm 的 state 并销毁这个对象。

看到这里，Vuex 的构造函数的介绍基本告一段落了，初始化核心就是 installModule 和 resetStoreVM 函数。通过对 mutations、actions 和 getters 的注册，我们了解到 state 的是按模块划分的，按模块的嵌套形成一颗状态树。而 actions、mutations 和 getters 的全局的，其中 actions 和 mutations 的 key 允许重复，但 getters 的 key 是不允许重复的。官方推荐我们给这些全局的对象在定义的时候加一个名称空间来避免命名冲突。

7.3.3　Vuex API 分析

Vuex 有一些常见的 API，例如 dispatch，commit，subscribe 等。接下来介绍一些关于 store 里的 API，一方面可以了解功能，另一方面可以借鉴一下他们的源码书写方式。

7.3.3.1　watch(getter, cb, options)

```
  watch (getter, cb, options) {
    assert(typeof getter === 'function', `store.watch only accepts a
function.`)
    return this._watcherVM.$watch(() => getter(this.state,
this.getters), cb, options)
  }
```

watch 作用是响应式的检测一个 getter 方法的返回值，因此函数首先断言 watch 的 getter 必须是一个函数，接着返回 this._watcherVM 的$watch 方法，观测 getter 方法返回值的变化，从而调用 cb 函数。

7.3.3.2 registerModule(path, rawModule)

```
registerModule (path, rawModule) {
    if (typeof path === 'string') path = [path]
    assert(Array.isArray(path), `module path must be a string or an
Array.`)
    this._modules.register(path, rawModule)
    installModule(this, this.state, path, this._modules.get(path))
    // reset store to update getters...
    resetStoreVM(this, this.state)
  }
```

顾名思义，这是个动态注册一个模块的 API，首先断言 path，path 必须符合 string 或者 array 的类型，接下来是注册的一套流程，在 this._modules 注册，调用 installModule 和 resetStoreVm 方法安装一遍。

7.3.3.3 unregisterModule (path)

```
unregisterModule (path) {
    if (typeof path === 'string') path = [path]
    assert(Array.isArray(path), `module path must be a string or an
Array.`)
    this._modules.unregister(path)
    this._withCommit(() => {
      const parentState = getNestedState(this.state, path.slice(0, -1))
      Vue.delete(parentState, path[path.length - 1])
    })
    resetStore(this)
  }
```

有注册模块就应该有注销模块，注销模块的代码结构跟注册模块的代码差不多，首先断言 path，在 this._moduls 注销，接下来通过_withCommit 进行删除当前模块的 state，最后通过 resetStore(this)完成注销。看如下 resetStore 的源码：

```
function resetStore (store, hot) {
  store._actions = Object.create(null)
  store._mutations = Object.create(null)
  store._wrappedGetters = Object.create(null)
  store._modulesNamespaceMap = Object.create(null)
  const state = store.state
  // init all modules
  installModule(store, state, [], store._modules.root, true)
```

```
    // reset vm
    resetStoreVM(store, state, hot)
  }
```

resetStore 的核心思想是重置 store 对象，然后再次安装一遍这个模块。这个方法里有第二个参数 hot，这里默认的是 true，此时我们再回到 installModule 方法里看有个 hot 属性，此次的 hot 为 true 传过去不会执行 state tree 的更新，表明此次安装是热更新，我们只需要重新注册一遍 mutations、actions 以及 getters 即可。

7.3.3.4　hotUpdate (newOptions)

```
  hotUpdate (newOptions) {
    this._modules.update(newOptions)
    resetStore(this, true)
  }
```

热更新方法，即根据传入参数更新模块 mutation、actions、getters，以及如果有子模块的话递归执行热更新函数，最后执行一遍 resetStore 更新方法。

7.3.3.5　replaceState (state)

```
  replaceState (state) {
    this._withCommit(() => {
      this._vm._data.$$state = state
    })
  }
```

替换 state 的 API 更为简单，只需要通过_withCommit 方法进行 state 替换即可。一般用于调试阶段。

7.3.4　辅助函数

Vuex 除了上述提供的 API 以外，还提供了一些辅助函数，目的是为了帮助我们使用 Vuex 的时候更方便，提供了操作 store 的各种属性的一系列语法糖，下面对它的概念介绍如下。

7.3.4.1　mapState

```
export const mapState = normalizeNamespace((namespace, states) => {
  const res = {}
```

```
    normalizeMap(states).forEach(({ key, val }) => {
      res[key] = function mappedState () {
        let state = this.$store.state
        let getters = this.$store.getters
        if (namespace) {
          const module = getModuleByNamespace(this.$store, 'mapState',
namespace)
          if (!module) {
            return
          }
          state = module.context.state
          getters = module.context.getters
        }
        return typeof val === 'function'
          ? val.call(this, state, getters)
          : state[val]
      }
      // mark Vuex getter for devtools
      res[key].Vuex = true
    })
    return res
  })
```

在介绍源码之前，简略地说明一些 mapState 的核心——将 store 中的 state 映射到 computed 的计算属性里。带着这句话我们来理解源码，源码一上来就有两个函数 normalizeNamespace 和 normalizeMap：

```
function normalizeMap (map) {
  return Array.isArray(map)
    ? map.map(key => ({ key, val: key }))
    : Object.keys(map).map(key => ({ key, val: map[key] }))
}

function normalizeNamespace (fn) {
  return (namespace, map) => {
    if (typeof namespace !== 'string') {
      map = namespace
      namespace = ''
    } else if (namespace.charAt(namespace.length - 1) !== '/') {
      namespace += '/'
    }
    return fn(namespace, map)
  }
}
```

normalizeMap 的定义是判定 map 是否为数组，若是数组，调用数组 map，将数组的值转化为({ key, val:key })对象，反之将 map 视为对象遍历 key，转化为({ key, val: map[key] })对象返回。

normalizeNamespace 函数返回一个函数，返回的函数接受两个参数并且对参数进行预处理的一个过程，最终会将处理后的参数传给定义的 fn 函数，进行核心逻辑处理。

调用 normalizeMap 之后将传入的 state 转化为{key, val}的对象数组，使用 forEach 遍历数组，构造一个新的对象，这个新对象每个元素都返回一个新的函数 mappedState，这个函数里首先获取全局的 state 和 getters，接着根据 namespace 获取当前的 module。如果获取成功，将 state 和 getters 进行重置为本模块，接着对 val 的类型判断，如果 val 是一个函数，则直接调用这个 val 函数，state 和 getters 作为参数，返回值作为 mappedState 的返回值；否则直接把 state[val]作为 mappedState 的返回值。配置完这个新的对象之后，还有一部 Vuex 属性设置，将其设置为 true，供在 devtools 里使用。

上面还有一个 getModuleByNamespace 的函数，该函数逻辑较为简单，通过 namespace 在 store._modulesNamespaceMap 属性里查找即可，查找不到进行错误提示，反之返回当前查找到的模块。

7.3.4.2　mapMutations

了解完 mapState 之后，对于 mapMutation 的理解就容易得多。由于源码解构上基本一致，我就贴出最核心、最不一样的代码进行解释，后面几个辅助函数相同。

```
  ...
    normalizeMap(mutations).forEach(({ key, val }) => {
      val = namespace + val
      res[key] = function mappedMutation (...args) {
       // 如果存在 namespace，那么对应的 module 也必须存在
        if (namespace && !getModuleByNamespace(this.$store,
'mapMutations', namespace)) {
          return
        }
       return this.$store.commit.apply(this.$store, [val].concat(args))
      }
    })
```

```
    return res
  ...
```

mapMutations 的核心是将 store 中的 commit 方法映射到组件的 methods 中，因此拿到 mapMutations 传入的数组或者对象后，同样构建一个对象，同样也是每个元素返回一个新的函数 mappedMutation，该函数最终返回全局的 this.$store.commit 方法，映射到 methods 的属性里。

7.3.4.3 mapActions

```
  ...
    normalizeMap(actions).forEach(({ key, val }) => {
      val = namespace + val
      res[key] = function mappedAction (...args) {
        if (namespace && !getModuleByNamespace(this.$store, 'mapActions',
namespace)) {
          return
        }
        return                    this.$store.dispatch.apply(this.$store,
[val].concat(args))
      }
    })
    return res
  ...
```

mapActions 和 mapMutations 极为相似，mapActions 的核心是将 store 中的 dispatch 方法映射到组件的 methods 中，因此拿到 mapActions 传入的数组或者对象后，同样构建一个对象，同样也是每个元素返回一个新的函数 mappedAction，该函数最终返回全局的 this.$store.dispatch 方法，映射到 methods 的属性里。

7.3.4.4 mapGetters

```
  ...
  normalizeMap(getters).forEach(({ key, val }) => {
    val = namespace + val
    res[key] = function mappedGetter () {
    // 如果存在 namespace，那么对应的 module 也必须存在
      if (namespace && !getModuleByNamespace(this.$store, 'mapGetters',
namespace)) {
        return
      }
```

```
      // 判定 getters 是否存在
      if (!(val in this.$store.getters)) {
        console.error(`[Vuex] unknown getter: ${val}`)
        return
      }
      return this.$store.getters[val]
    }
    // mark Vuex getter for devtools
    res[key].Vuex = true
  })
  return res
...
```

mapGetters 和 mapState 极为相似，mapGetters 的核心是将 store 中的 getters 方法映射到组件的 computed 中，因此拿到 mapGetters 传入的数组或者对象后，同样构建一个对象，同样也是每个元素返回一个新的函数 mappedGetter，该函数最终返回全局的 this.$store.getters 方法，映射到 computed 的计算属性里。

7.3.5　插件

7.3.5.1　插件使用

```
plugins.concat(devtoolPlugin).forEach(plugin => plugin(this))
```

7.3.5.2　两个插件

devtoolPlugin

devtoolPlugin

```
const devtoolHook =
  typeof window !== 'undefined' &&
  window.__Vue_DEVTOOLS_GLOBAL_HOOK__

export default function devtoolPlugin (store) {
  if (!devtoolHook) return

  store._devtoolHook = devtoolHook

  devtoolHook.emit('Vuex:init', store)
```

```
    devtoolHook.on('Vuex:travel-to-state', targetState => {
      store.replaceState(targetState)
    })

    store.subscribe((mutation, state) => {
      devtoolHook.emit('Vuex:mutation', mutation, state)
    })
  }
```

从插件对外暴露的函数来看，函数首先判断 devtoolHook 的值，如果浏览器安装了 Vue 开发者工具，那么 window 上就会有一个_Vue_DEVTOOLS_GOLBAL_HOOK_的引用，devtoolHook 就是指向这个引用的。然后通过派发一个 Vuex:init，让开发者工具拿到这个 store 实例。通过绑定监听 travel-to-state 的事件，把当前的状态树替换为目标树，这个功能是利用 Vue 开发者工具替换 Vuex 的状态。最后通过 subscribe 订阅 store 的 state 的变化，store 的 mutation 提交了 state 的变化，会触发回调函数——通过派发一个 Vuex:mutation 事件，mutation 和 state 作为参数，这样开发者工具可以实时展示最新的状态树。

loggerPlugin

```
  import { deepCopy } from '../util'

  export default function createLogger ({
    collapsed = true,
    transformer = state => state,
    mutationTransformer = mut => mut
  } = {}) {
    return store => {
      let prevState = deepCopy(store.state)

      store.subscribe((mutation, state) => {
        if (typeof console === 'undefined') {
          return
        }
        const nextState = deepCopy(state)
        const time = new Date()
        const    formattedTime    =    `    @    ${pad(time.getHours(),
2)}:${pad(time.getMinutes(),                2)}:${pad(time.getSeconds(),
2)}.${pad(time.getMilliseconds(), 3)}`
        const formattedMutation = mutationTransformer(mutation)
        const message = `mutation ${mutation.type}${formattedTime}`
        const startMessage = collapsed
```

```
        ? console.groupCollapsed
        : console.group

      // render
      try {
        startMessage.call(console, message)
      } catch (e) {
        console.log(message)
      }

      console.log('%c prev state', 'color: #9E9E9E; font-weight: bold',
transformer(prevState))
      console.log('%c mutation', 'color: #03A9F4; font-weight: bold',
formattedMutation)
      console.log('%c next state', 'color: #4CAF50; font-weight: bold',
transformer(nextState))

      try {
        console.groupEnd()
      } catch (e) {
        console.log('— log end —')
      }

      prevState = nextState
    })
  }
}

function repeat (str, times) {
  return (new Array(times + 1)).join(str)
}

function pad (num, maxLength) {
  return repeat('0', maxLength - num.toString().length) + num
}
```

logger 插件是把 mutation 的动作以及 store 的 state 变化实时输出，logger 插件对外暴露了 createLogger 方法，方法返回一个函数。该函数首先通过 deepCopy 进行保存当前的 state 为 prevState，接着 store 订阅 state 的变化，回调函数里会保存变化后的 state 为 nextState，然后进行一系列的格式化数据处理，再通过 console.log 输出显示。在函数的最后，我们把 nextState 赋值给 prevState，便于下一次的 mutation 动作输出。

7.3.6 一些函数的封装

这部分会将整个源码结构里一些公共函数独立出来进行分析，一方面说明一些函数逻辑，另一方面可以借鉴一些书写方式和设计思路。

7.3.6.1 deepCopy

```
export function deepCopy (obj, cache = []) {
  if (obj === null || typeof obj !== 'object') {
    return obj
  }

  const hit = find(cache, c => c.original === obj)
  if (hit) {
    return hit.copy
  }

  const copy = Array.isArray(obj) ? [] : {}
  cache.push({
    original: obj,
    copy
  })

  Object.keys(obj).forEach(key => {
    copy[key] = deepCopy(obj[key], cache)
  })

  return copy
}
```

深度拷贝函数存在的意义是当需要拷贝的为一个对象时，我们需要逐层渗透拷贝，直到完全拷贝，该函数里还使用了缓存机制。

7.3.6.2 getNestedState

```
function getNestedState (state, path) {
  return path.length
    ? path.reduce((state, key) => state[key], state)
    : state
}
```

该方法是查找当前 path 对应的 state，巧妙地使用 reduce 方法让代码看起来很简洁又实用。

7.3.6.3　unifyObjectStyle

```
function unifyObjectStyle (type, payload, options) {
  if (isObject(type) && type.type) {
    options = payload
    payload = type
    type = type.type
  }

  assert(typeof type === 'string', `Expects string as the type, but found ${typeof type}.`)

  return { type, payload, options }
}
```

unifyObjectStyle 方法是对传出参数的一次统一兼容处理，因为在使用 Vuex 的过程中我们可以在 type 属性上使用对象。

7.3.6.4　isPromise & assert

```
export function isPromise (val) {
  return val && typeof val.then === 'function'
}

export function assert (condition, msg) {
  if (!condition) throw new Error(`[Vuex] ${msg}`)
}
```

两个简单判定函数，isPromise 函数是用来检测是否支持 promise 的执行，直接通过获取值的 then 属性进行判定即可，assert 是个断言函数。

这一节介绍了 Vuex2.0 的源码，整个源码量不大，通过库核心介绍、周边 API 和辅助函数多个方面去理解源码。对于一个库和源码研究，研究前和研究后一定要反复使用这个库，阅读前使用库帮助我们了解这是个什么库，可以做什么；阅读后使用库可以让我们从更多细节上去推敲使用上的细节，以及为什么这么使用，后者有没有更好的使用方式。

虽然源码读起来晦涩难懂，但是，当你通读完以后，可以帮助你了解内部的

机制，从而更好地使用 Vuex，也更容易帮助你进行 debug；其次通过通读源码，整体地理解它的设计理念以及编码风格，帮助你日后在迈向技术高工路上进行实践学习。

7.4 Vue-router 原理

7.4.1 Vue-router

Vue-router 是一个专为 Vue 开发的路由管理模式，主要用来构建单页面应用。
 Vue 的单页应用基于路由和组件，路由用于设定访问路径，并将路径和组件映射起来。

7.4.2 Vue-router 应用举例

```
import Vue from 'vue'
import VueRouter from 'vue-router'

// 1. 插件.
// 安装<router-view>和<router-link>两个组件,
// 给当前应用下所有的组件注入$router and $route 两个属性
Vue.use(VueRouter)

// 2. 定义各个路由下使用的组件，简称路由组件
//   每个路由应该映射一个组件。其中"component" 可以是
//    通过 Vue.extend() 创建的组件构造器,
//    或者，只是一个组件配置对象。
const Home = { template: '<div>home</div>' }
const Foo = { template: '<div>foo</div>' }
const Bar = { template: '<div>bar</div>' }

// 3. 创建 VueRouter 实例 router,, 然后传`routes`配置
const router = new VueRouter({
  mode: 'history',
  base: __dirname,
  routes: [
    { path: '/', component: Home },
    { path: '/foo', component: Foo },
    { path: '/bar', component: Bar }
  ]
})
```

```
  // 4. 创建启动应用
  // 一定要确认注入了 router,通过 router 配置参数注入路由,从而让整个应用都有路由功
能。
  // 在 <router-view> 中将会渲染路由组件
  new Vue({
    router,
    template: `
      <div id="app">
        <h1>Basic</h1>
        <ul>
          <li><router-link to="/">/</router-link></li>
          <li><router-link to="/foo">/foo</router-link></li>
          <li><router-link to="/bar">/bar</router-link></li>
          <router-link    tag="li"    to="/bar"    :event="['mousedown',
'touchstart']">
            <a>/bar</a>
          </router-link>
        </ul>
        <router-view class="view"></router-view>
      </div>
    `
  }).$mount('#app')
```

7.4.3　Vue-router 原理

7.4.3.1　目录结构

目录结构如图 7-10 所示。

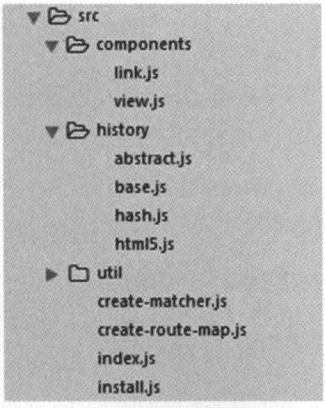


图 7-10

整个目录结构非常简单，其中 components 目录是 router 相关的两个组件，history 目录是针对不同的模式进行路由处理，主要包括 HashHistory、HtmlHistory、AbstractHistory 三种 history 模式。

util 目录是抽象出的一些公共方法。

create-matcher.js 主要用于 match 匹配函数。

create-router-map.js 主要用于根据用户路由配置对象生成普通的 path 来对应的路由记录以及根据 name 来对应的路由记录 map。

index.js 主要是路由的初始化。

install.js 主要用于注入全局的 Vue 对象属性。

和流程相关的主要关注点就是 components、history 目录以及 create-matcher.js、create-route-map.js、index.js、install.js。

7.4.3.2 安装方法

Vue-router 本身是一个库，需要接入 Vue 的环境才能使用。

Vue.use(VueRouter)的执行会调用 install 方法，它的核心内容是注入$router 和 $router 两个属性。

$router 代表 router 实例。

$route 代表当前激活的路由信息对象。

vue-router 的安装源码如下：

```
export function install (Vue) {

  if (install.installed) return

  install.installed = true
  //赋值私有的 Vue 引用
   _Vue = Vue
  //注入$router,$route
  Object.defineProperty(Vue.prototype, '$router', {
    get () { return this.$root._router }
  })

  Object.defineProperty(Vue.prototype, '$route', {
    get () { return this.$root._route }
  })

  const isDef = v => v !== undefined
```

```
    const registerInstance = (vm, callVal) => {
      let i = vm.$options._parentVnode
      if (isDef(i) && isDef(i = i.data) && isDef(i =
i.registerRouteInstance)) {
        i(vm, callVal)
      }
    }

    Vue.mixin({
      beforeCreate () {
        //响应式的_route 属性，属性值发生变化，就去触发更新机制
        if (isDef(this.$options.router)) {
          this._router = this.$options.router
          this._router.init(this)
          Vue.util.defineReactive(this, '_route', this._router.history.
current)
        }
        registerInstance(this, this)
      },
      destroyed () {
        registerInstance(this)
      }
    })
    //组件
    Vue.component('router-view', View)
    Vue.component('router-link', Link)

    const strats = Vue.config.optionMergeStrategies
    // use the same hook merging strategy for route hooks
    strats.beforeRouteEnter = strats.beforeRouteLeave = strats.created
  }
```

首先判断插件是否已经安装过，如果已经安装过则不用重新安装。

安装过程：

（1） export 是 Vue 的一个应用，在 install 方法中赋值私有的_Vue 应用，便于其他地方使用 Vue 的方法而不用重新引用。

（2） 通过给 Vue.prototype 定义 $router、$route 属性，可以把它们注入到所有的组件中。

（3） Vue.mixin 混合 beforeCreate 方法用在 new Vue()的时候调用。

（4） 注册全局组件 router-view 和 router-link。

7.4.3.3 初始化

实例化 VueRouter：

```
// 创建 VueRouter 实例 router
const router = new VueRouter({
  mode: 'history',
  base: __dirname,
  routes: [
    { path: '/', component: Home },
    { path: '/foo', component: Foo },
    { path: '/bar', component: Bar }
  ]
})
```

（1）初始化 app，赋值 options，注册导航钩子。

（2）创建 match 匹配函数，根据 options.routes 对象创建 match。

```
传入 routes 参数，创建路由 map
```

create-matcher

创建路由 map、addRoutes 的方法，match 匹配函数方法。

create-route-map

遍历 routes 对象，调用 addRouteRecord 方法，创建 RouteRecord 对象，返回 pathList、pathMap、nameMap：

```
addRouteRecord 方法
  标准化 path
  RouteRecord 对象包括
  判断是否有 children
  判断是否有 alias 参数
```

（3）初始化 history 对象根据传入的 mode 参数来判断实例化的 History 对象。mode = "hash" | "history" | "abstract"。

HashHistory：

使用 URL hash 值来作路由。支持所有浏览器，包括不支持 HTML5 History API 的浏览器使用路由的 hash 模式来模拟一个完整的 url，当 url 改变时，不会重新加载页面。

HTML5History：

依赖 HTML5 History API 和服务器配置。

mode:'history'这种模式充分利用 history.pushState()API 来完成 URL 跳转而无须重新加载页面。

参考<a>https://developer.mozilla.org/en-US/docs/Web/API/History_API

HTML5 新增 history.pushState()和 history.replaceState()的方法：

使用 history.pushState()在更改状态后创建的 XMLHttpRequest 对象的 HTTP 头中使用的引用。这个引用是创建 XMLHttpRequest 对象时，document 的 URL。

当 history 实体被改变时，popstate 事件将会发生。如果 history 实体是由 pushState 和 replaceState 方法产生的，popstate 事件的 state 属性会包含一份来自 history 实体的 state 对象的拷贝。

当页面加载时，它可能会有一个非空的 state 对象。这可能发生在当页面设置一个 state 对象（使用 pushState 或者 replaceState）之后用户重启了浏览器。当页面重新加载，页面将收到 onload 事件，但不会有 popstate 事件。然而，如果你读取 history.state 属性，将在 popstate 事件发生后得到这个 state 对象。

AbstractHistory：

支持所有 JavaScript 的运行环境，如 Node.js 服务器端。如果发现没有浏览器的 API，路由会自动强制进入这个模式。

7.4.3.4　实例化 History 对象

```
switch (mode) {
  case 'history':
    this.history = new HTML5History(this, options.base)
    break
  case 'hash':
    this.history = new HashHistory(this, options.base, this.fallback)
    break
  case 'abstract':
    this.history = new AbstractHistory(this, options.base)
    break
  default:
    if (process.env.NODE_ENV !== 'production') {
      assert(false, `invalid mode: ${mode}`)
    }
  }}
```

构造函数。

实例化 HTML5History，需要传递参数 router（当前 router 对象），base（应用的基本路径）

7.4.3.5 实例化 Vue

new Vue()会调用 Vue 生命周期的 beforeCreate 方法。

```
  beforeCreate () {
      //响应式的_route 属性，属性值发生变化，就去触发更新机制
      if (isDef(this.$options.router)) {
        this._router = this.$options.router
        this._router.init(this)
        Vue.util.defineReactive(this, '_route', this._router.history.
current)
      }
      registerInstance(this, this)
    }
```

调用 VueRouter 的 init 的方法：

初始化 app；

执行 history 的 transitionTo 方法；

如果是 HashHisitory，要先监听 hashchange 事件。

```
  init (app: any /* Vue component instance */) {
    process.env.NODE_ENV !== 'production' && assert(
      install.installed,
      `not installed. Make sure to call \`Vue.use(VueRouter)\` ` +
      `before creating root instance.`
    )

    this.apps.push(app)

    // main app already initialized.
    if (this.app) {
      return
    }

    this.app = app

    const history = this.history
```

```
    if (history instanceof HTML5History) {
      history.transitionTo(history.getCurrentLocation())
    } else if (history instanceof HashHistory) {
      const setupHashListener = () => {
        history.setupListeners()
      }
      history.transitionTo(
        history.getCurrentLocation(),
        setupHashListener,
        setupHashListener
      )
    }
    //设置当前历史对象的 cb 值
    history.listen(route => {
      this.apps.forEach((app) => {
        app._route = route
      })
    })
  }
```

transitionTo 方法：

调用 match 得到匹配的 route 对象。

match 方法是在 create-matcher.js 中定义的。

```
  transitionTo (location: RawLocation, onComplete?: Function, onAbort?:
Function) {
    //调用 match 得到匹配的 route 对象
    const route = this.router.match(location, this.current)
    //确认过渡
    this.confirmTransition(route, () => {
      //更新当前 route 对象
      this.updateRoute(route)
      onComplete && onComplete(route)
      //子类实现的更新 url 地址
      // 对于 hash 模式的话 就是更新 hash 的值
      // 对于 history 模式的话 就是利用 pushstate / replacestate 来更新
      // 浏览器地址
      this.ensureURL()

      // fire ready cbs once
      if (!this.ready) {
        this.ready = true
        this.readyCbs.forEach(cb => { cb(route) })
      }
```

```
    }, err => {
      if (onAbort) {
        onAbort(err)
      }
      if (err && !this.ready) {
        this.ready = true
        this.readyErrorCbs.forEach(cb => { cb(err) })
      }
    })
  }
```

normalizeLocation，获取标准化的 location；

获取 location 对象；

遍历 pathList 数组判断当前的 location 的 path 是否匹配 record.regex 正值表达式。

如果返回找到匹配的 location.path，调用_createRoute 方法创建路由：

```
  function match (
    raw: RawLocation,
    currentRoute?: Route,
    redirectedFrom?: Location
  ): Route {
    const location = normalizeLocation(raw, currentRoute, false, router)
    const { name } = location
    //命名路由处理
    if (name) {
      //nameMap[name] = 路由记录
      const record = nameMap[name]
      if (process.env.NODE_ENV !== 'production') {
        warn(record, `Route with name '${name}' does not exist`)
      }
      const paramNames = record.regex.keys
        .filter(key => !key.optional)
        .map(key => key.name)

      if (typeof location.params !== 'object') {
        location.params = {}
      }

      if (currentRoute && typeof currentRoute.params === 'object') {
        for (const key in currentRoute.params) {
          if (!(key in location.params) && paramNames.indexOf(key) > -1)
{
```

```
            location.params[key] = currentRoute.params[key]
          }
        }
      }

      if (record) {
        location.path = fillParams(record.path, location.params, `named
route "${name}"`)
      //创建 route
        return _createRoute(record, location, redirectedFrom)
      }
    } else if (location.path) {
     //普通路由处理
      location.params = {}
      for (let i = 0; i < pathList.length; i++) {
        const path = pathList[i]
    //pathMap[path] = 路由记录
        const record = pathMap[path]
        if (matchRoute(record.regex, location.path, location.params)) {
       //匹配成功 创建 route
          return _createRoute(record, location, redirectedFrom)
        }
      }
    }
    // no match
    return _createRoute(null, location)
  }

  function _createRoute (
    record: ?RouteRecord,
    location: Location,
    redirectedFrom?: Location
  ): Route {
   //重定向和别名逻辑
    if (record && record.redirect) {
      return redirect(record, redirectedFrom || location)
    }
    if (record && record.matchAs) {
      return alias(record, location, record.matchAs)
    }
    return createRoute(record, location, redirectedFrom, router)
  }
```

7.4.3.6 view 渲染逻辑

目录：src/components/view.js。

render 方法，渲染界面：

```
1. 解决嵌套深度问题
data.routerView = true
2. route 对象
const route = parent.$route
3.缓存
const cache = parent._routerViewCache || (parent._routerViewCache = {})
4.获取 depth-当前组件的深度
5.keep-alive 逻辑处理
如果 inactive 为 true，直接读取 cache 中相应层级的记录，如果为 false，执行 6
6.matched
得到相匹配的当前组件层级的路由记录
7.得到要渲染的而组件。
8.调用 createElement 函数，渲染匹配的组件
```

7.4.3.7 link 渲染逻辑

src/components/link.js。

render 方法：

```
1.得到 router 实例以及当前激活的 route 对象
    const router = this.$router<br>
    const current = this.$route<br>
2.router.resolve 调用 index.js 里面的 resolve 方法
 normalizeLocation 标准化 path
 获取匹配到的 router 对象
 根据当前目标链接和当前激活的 route 匹配结果
 const{location,route,href}   =   router.resolve(this.to,   current,
this.append)
 const route = this.match(location, current)
3.根据 path 和 base 生成 href
   最后返回 loaction，router，href 对象
4.激活 class 优先当前组件上获取 要么就是 router 配置的 linkActiveClass
   默认 router-link-active
5.相比较目标<br>
   因为有命名路由 所以不一定有 path
6.如果严格模式的话 就判断是否是相同路由（path query params hash）
   否则就走包含逻辑（path 包含，query 包含 hash 为空或者相同）
```

```
7.事件绑定
8.创建元素
```

可以看出 router-link 组件就是在其点击的时候根据设置的 to 的值去调用 router 的 push 或者 replace 来更新路由的，同时呢，会检查自身是否和当前路由匹配（严格匹配和包含匹配）来决定自身的 activeClass 是否添加。

本节主要介绍 Vue-router 的原理，从 Vue-router 的初始化入手，通过 API 文档等深入理解源码。通过阅读源码，更好地了解内部的实现机制，对于自己的编码风格也有好的借鉴作用。

读 者 圈

第8章　进军WEEX

8.1　搭建WEEX基础环境

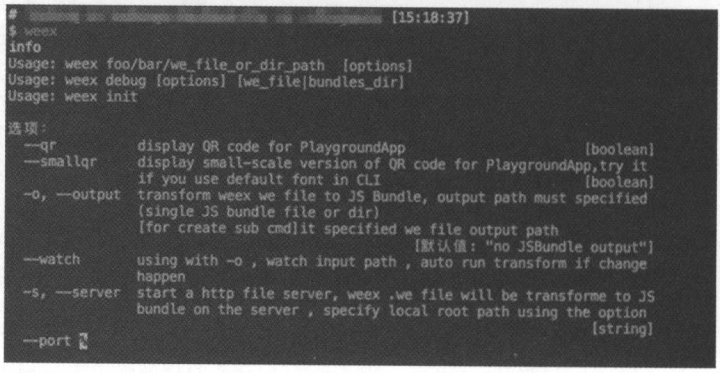
在开发WEEX应用前，我们需要搭建起WEEX的基础环境，这里面包括Node、npm、WEEX-toolkit、playground等一系列工具。

首先Node和npm在本书前面的部分已经提到过了，这里就不再重复，但是需要注意的是，我们安装的Node版本不能比WEEX官网推荐的版本低，否则可能会导致报错。

WEEX-toolkit仅有最新的beta版本开始才支持初始化Vue项目，使用前请确认版本是否正确。

```
$ npm install -g WEEX-toolkit@beta
```

安装结束后你可以直接使用WEEX命令验证是否安装成功，它会显示WEEX命令行工具各参数：



8.1.1　初始化：hello world

首先初始化Weex项目：

```
$ WEEX init awesome-project
```

执行完命令后，在 awesome-project 目录中创建一个使用 Weex 和 Vue 的模板项目。

之后我们进入项目所在路径，WEEX-toolkit 已经为我们生成了标准项目结构。

在 package.json 中，已经配置好了几个常用的 npm script，分别介绍如下。

- build：源码打包，生成 JS Bundle；
- dev：webpack watch 模式，方便开发；
- serve：开启静态服务器；
- debug：调试模式。

我们先通过 npm install 安装项目依赖。之后运行 npm run dev 和 npm run serve，开启 watch 模式和静态服务器。

然后我们打开浏览器，进入 http://localhost:8080/index.html 即可看到 WEEX h5 页面。

初始化时已经为我们创建了基本的示例，我们可以在 src/foo.vue 中查看。

8.1.2　dotwe

向对 WEEX 感兴趣的人推荐 dotwe.org 这个网站，这个网站是 WEEX 官方推荐的 WEEX 在线开发网站，搭配 playground app，我们能快速地体验 WEEX 的开发效果。如图 8-1 所示。

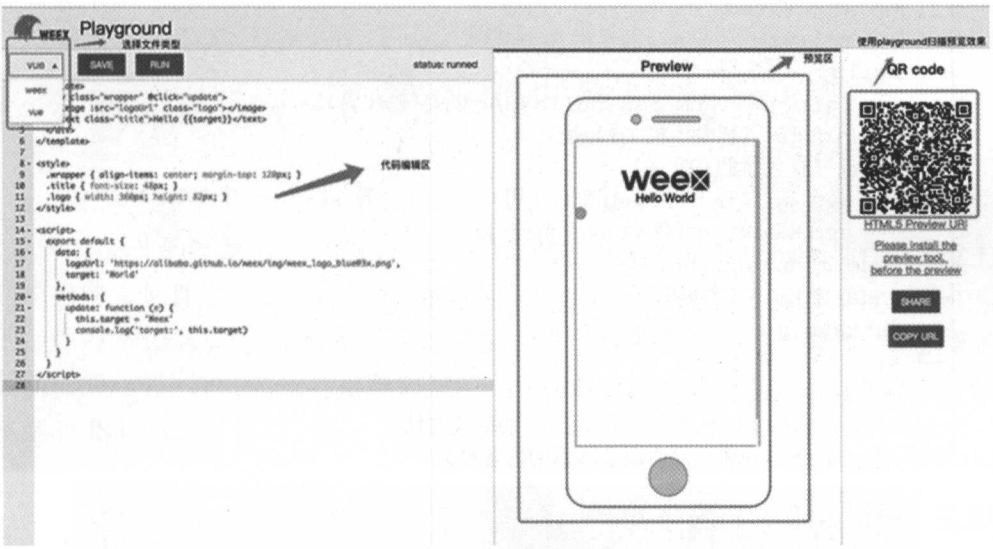


图 8-1

当我们编辑完代码之后，点击 save 按钮我们的网页链接将会发生改变，我们能够通过分享该链接给别人来预览页面。同时这个链接能够用在 github 的 issue 上，作为复现问题的一个快捷便利的环境。

点击 run 后，在预览区会自动展示页面效果，同时二维码区域会生成供 playground App 扫描的二维码。

使用 dotWe 对 WEEX 尝鲜是一个不错的选择，如果你想更专业的开发 WEEX，还需要搭建 WEEX 的开发环境。

8.2 分析首个 WEEX 工程代码

8.2.1 目录结构

我们先来看一下 WEEX 自动生成项目的目录结构（将 node_modules 中的目录过滤掉后）：

```
├── README.md （说明文件）
├── app.js （入口文件）
├── assets （一些 weex.html 中引用的资源）
│   ├── phantom-limb.js （用来模拟移动 touch 点击效果的库）
│   ├── qrcode.js （主页面中引用来生成二维码的库）
│   ├── qrcode.min.js
│   ├── style.css
│   └── url.js
├── build
│   └── init.js （npm run build 中用到的打包代码）
├── config.js （记录本机 Ip 地址）
├── dist （打包后的文件夹）
│   ├── app.web.js （供 web 平台使用）
│   └── app.weex.js（供 weex 平台使用）
├── node_modules （npm 包）
├── index.html （主页面）
├── package.json
├── src
│   └── foo.vue
├── webpack.config.js （webpack 的配置文件）
└── weex.html （主页面中嵌套展示的内容页）
```

8.2.2 通过 serve 起服务

npm run serve 启用了 serve 库，这样我们能够通过 ip+端口的形式获取到

index.html、weex.html 和 dist 中的 app.web.js。如图 8-2 所示。

图 8-2

8.2.3　webpack 配置

1.　webpack banner 配置

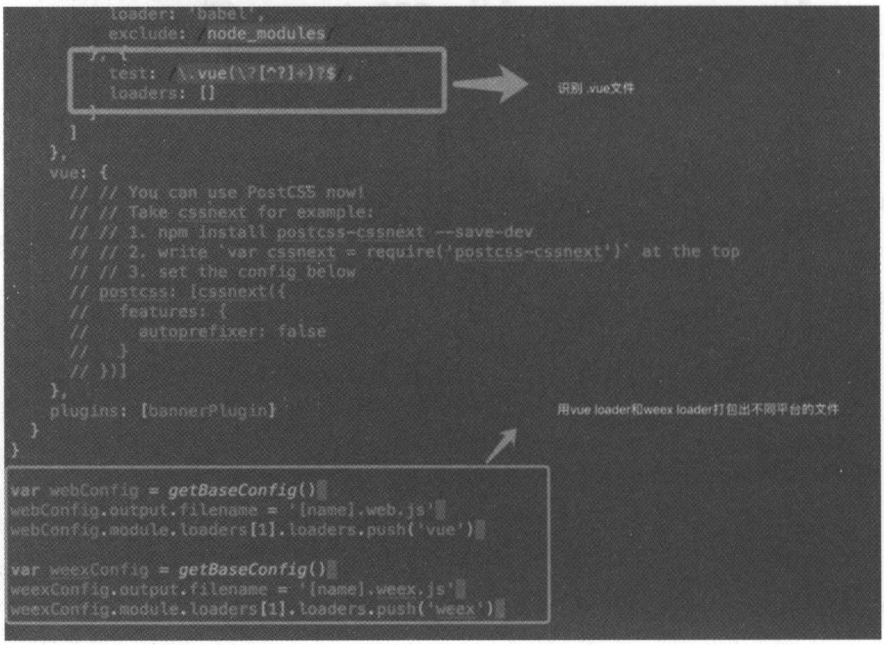
webpack 的配置文件则比较简单，就是指定了 webpack banner 插件在每个打包出来的 js 文件中生成 Vue frame 的声明，这个声明是提供给 WEEX js 引擎识别的，因为 WEEX 的 js 引擎不只支持 Vue 这个前端框架，它还支持你自己扩展的其他框架，同时 WEEX 支持多种框架在一个移动应用中共存并各自解析基于不同框架的 JS bundle。

```
// { "framework": "Vue" }

/******/ (function(modules) { // webpackBootstrap
```

生成 Vue frame 的声明如下：

```
var bannerPlugin = new webpack.BannerPlugin(
  '// { "framework": "Vue" }\n',
  { raw: true }
)
```

2. webpack loader 配置



8.2.4 页面开发

打包的入口文件是 app.js，src 下的 foo.Vue 可以看作是页面文件，在这里我们可以用熟悉的前端开发模式去开发我们的页面了。

8.3 debug WEEX 代码

8.3.1 web 端调试

通过 npm run serve 和 npm run dev 起好服务后，我们通过范围 http://

localhost:8080/WEEX.html 这个链接来调试我们的网页，如图 8-3 所示。

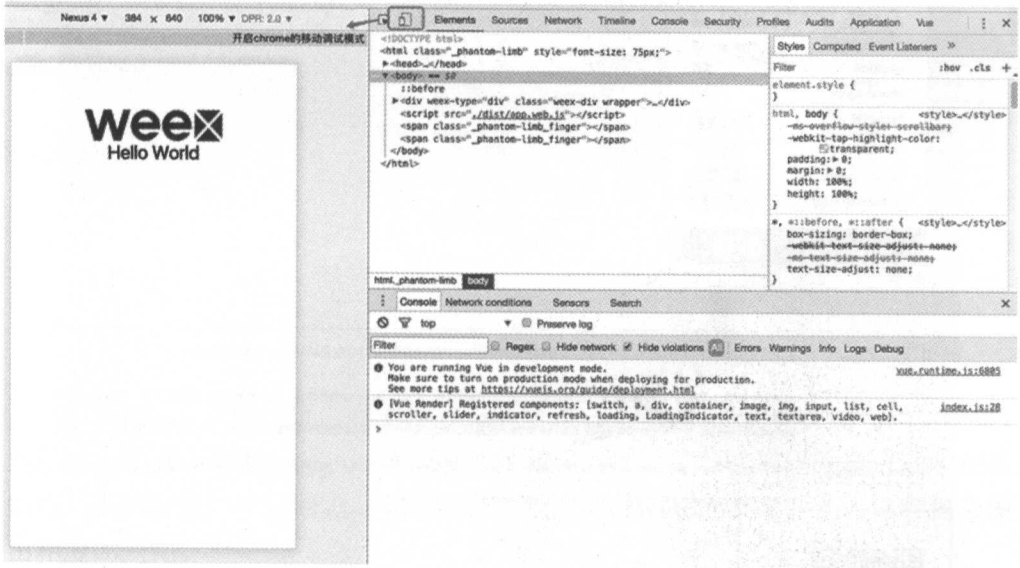


图 8-3

通过开启 chrome 的移动调试模式，我们能够使用熟悉的调试手段来调试我们的页面，同时 WEEX.html 代码里的 WEEX-Vue-render/index.js 会在页面上注册 WEEX 自带的组件。

```
<script src="./dist/app.web.js"></script>
<script src="./dist/index.web.js"></script>
<script src="./node_modules/weex-vue-render/index.js"></script>
```

8.3.2　手机端调试

只在 Web 端调试页面是不能发现所有问题的，接下来为大家介绍 WEEX 提供的手机端调试方法。

我们前面已经安装好了 WEEX-toolkit，现在我们在项目目录下运行：

```
weex debug
```

WEEX-toolkit 会帮助我们起好 debug serve 服务，如图 8-4 所示。

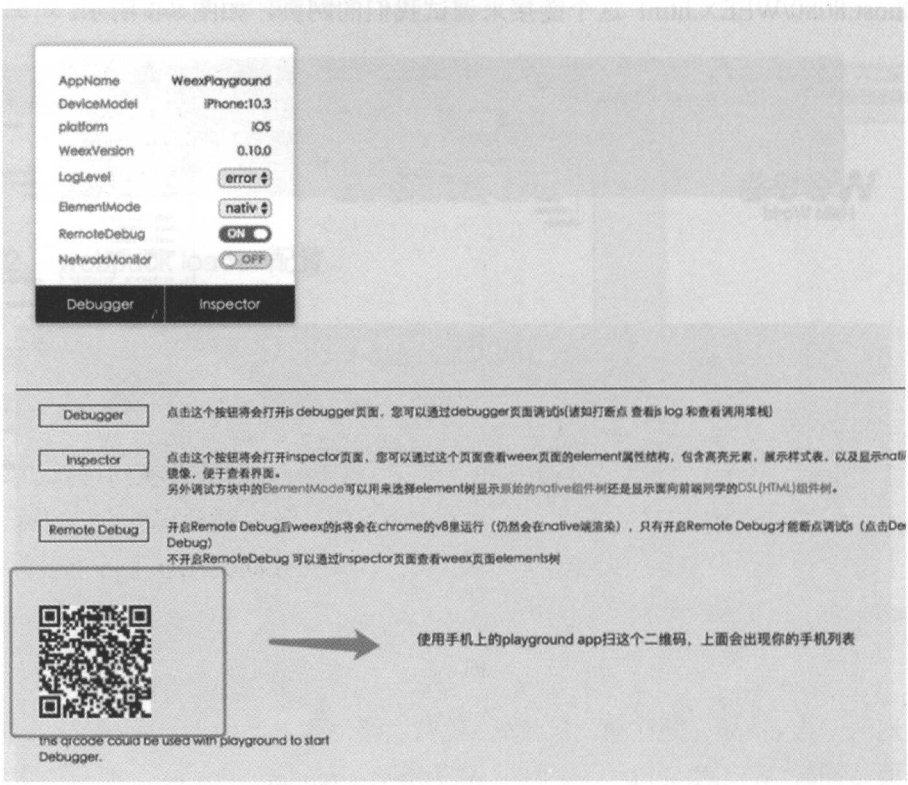


图 8-4

同时我们还可以用手机再去扫之前我们在项目中用 npm run serve 起的项目二维码，这个时候手机上就会打开我们要调试的页面了。由于 playground App 仍然不够完善，所以，有时候并不会一次成功，我们重启 playground App 再试一下就可以了。

回到我们的 debub 页面，机器列表上有 Debugger 和 Inspector 两个按钮，点击 Inspector 按钮将会打开 inspector 页面，可以通过这个页面查看 WEEX 页面的 element 属性结构，包含高亮元素、展示样式表，以及显示 native 的 log。另外，调试方块中的 ElementMode，可以用来选择 element 树，显示原始的 native 组件树或显示面向前端同学的 DSL（HTML）组件树。

点击 debugger 按钮，可以通过 debugger 页面调试 js（诸如打断点，查看 js log 和查看调用堆栈）。如图 8-5 所示。

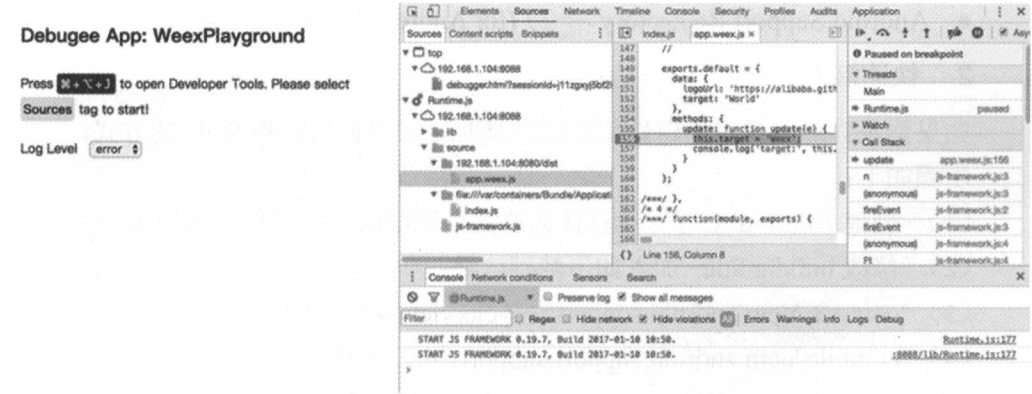


图 8-5

按照图 8-5 中的位置，我们在 chrome、devtool 的 source 页找到代码的位置，然后打上断点，点击 playground App 上的相应区域，这时候代码就会自动断在我们打断点的位置。跳过断点，代码会自动执行，playground App 上的内容也会做出相应的变化。

8.4　集成 WEEX 到已有应用

8.4.1　集成到 Android

注意：以下文档都是假设已经具备一定的 Android 开发经验的人员适用。

8.4.1.1　Android 集成有两种方式

（1）源码依赖：能够快速使用 WEEX 最新功能，可以根据自己项目的特性进行相关改进。

（2）SDK 依赖：WEEX 会在 jcenter 定期发布稳定版本。

注意：国内可能需要翻墙。

1.　前期准备

- 已经安装了 JDK version>=1.7，并配置了环境变量；
- 已经安装了 Android SDK，并配置环境变量；
- Android SDK version 23 (compileSdkVersion in build.gradle)；
- SDK build tools version 23.0.1 (buildToolsVersion in build.gradle)；

- Android Support Repository >= 17 (for Android Support Library)。

2. 快速接入

如果你是尝鲜或者对稳定性要求比较高时，可以使用依赖 SDK 的方式。步骤如下：

（1） 创建 Android 工程，没有什么要特别说明的，按照你的习惯来做；

（2） 修改 build.gradle 加入如下基础依赖；

（3） compile 'com.android.support:recyclerview-v7:23.1.1';

（4） compile 'com.android.support:support-v4:23.1.1';

（5） compile 'com.android.support:appcompat-v7:23.1.1';

（6） compile 'com.alibaba:fastjson:1.1.46.android'。

compile 'com.taobao.android:weex_sdk:0.5.1@aar'

注意：版本可以高不可以低。

3. 代码实现

注意：附录中有完整代码地址。

- 实现图片下载接口，初始化时设置。

```
package com.weex.sample;

import android.widget.ImageView;

import com.taobao.weex.adapter.IWXImgLoaderAdapter;
import com.taobao.weex.common.WXImageStrategy;
import com.taobao.weex.dom.WXImageQuality;

/**
 * Created by lixinke on 16/6/1.
 */
public class ImageAdapter implements IWXImgLoaderAdapter {


    @Override
    public void setImage(String url, ImageView view, WXImageQuality
quality, WXImageStrategy strategy) {
        //实现你自己的图片下载，否则图片无法显示。
    }
}
```

- 初始化。

```
package com.weex.sample;

import android.app.Application;

import com.taobao.weex.InitConfig;
import com.taobao.weex.WXSDKEngine;

/**
 * 注意要在 Manifest 中设置 android:name=".WXApplication"
 * 要实现 ImageAdapter 否则图片不能下载
 * gradle 中一定要添加一些依赖，否则初始化会失败。
 * compile 'com.android.support:recyclerview-v7:23.1.1'
 * compile 'com.android.support:support-v4:23.1.1'
 * compile 'com.android.support:appcompat-v7:23.1.1'
 * compile 'com.alibaba:fastjson:1.1.45'
 */
public class WXApplication extends Application {

  @Override
  public void onCreate() {
    super.onCreate();
    InitConfig    config=new    InitConfig.Builder().setImgAdapter(new
ImageAdapter()).build();
    WXSDKEngine.initialize(this,config);
  }
}
```

- 开始渲染。

```
package com.weex.sample;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;

import com.taobao.weex.IWXRenderListener;
import com.taobao.weex.WXSDKInstance;
import com.taobao.weex.common.WXRenderStrategy;
import com.taobao.weex.utils.WXFileUtils;
```

```
public class MainActivity extends AppCompatActivity implements
IWXRenderListener {

    WXSDKInstance mWXSDKInstance;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      mWXSDKInstance = new WXSDKInstance(this);
      mWXSDKInstance.registerRenderListener(this);
      /**
       * WXSample 可以替换成自定义的字符串，针对埋点有效。
       * template 是.we transform 后的 js 文件。
       * option 可以为空，或者通过 option 传入 js 需要的参数。例如 bundle js 的地
址等。
       * jsonInitData 可以为空。
       * width 为-1 默认全屏，可以自己定制。
       * height =-1 默认全屏，可以自己定制。
       */
      mWXSDKInstance.render("WXSample",
WXFileUtils.loadFileContent("hello.js", this), null, null, -1, -1,
WXRenderStrategy.APPEND_ASYNC);
    }

    @Override
    public void onViewCreated(WXSDKInstance instance, View view) {
      setContentView(view);
    }

    @Override
    public void onRenderSuccess(WXSDKInstance instance, int width, int
height) {

    }

    @Override
    public void onRefreshSuccess(WXSDKInstance instance, int width, int
height) {

    }
```

```
    @Override
    public void onException(WXSDKInstance instance, String errCode,
String msg) {

    }

    @Override
    protected void onResume() {
      super.onResume();
      if(mWXSDKInstance!=null){
        mWXSDKInstance.onActivityResume();
      }
    }

    @Override
    protected void onPause() {
      super.onPause();
      if(mWXSDKInstance!=null){
        mWXSDKInstance.onActivityPause();
      }
    }

    @Override
    protected void onStop() {
      super.onStop();
      if(mWXSDKInstance!=null){
        mWXSDKInstance.onActivityStop();
      }
    }

    @Override
    protected void onDestroy() {
      super.onDestroy();
      if(mWXSDKInstance!=null){
        mWXSDKInstance.onActivityDestroy();
      }
    }
  }
```

8.4.1.2 源码依赖（IDE Android Studio）

（1） 下载源码 git clone https://github.com/alibaba/weex。

（2） 创建 Android 工程。

（3） 通过相关路径引入 SDK Module File->New-Import Module->，选择 WEEX SDK Module(weex/android/sdk) -> Finish。

（4） 在 App 的 build.gradle 中添加如下依赖:compile project(':weex_sdk')。

（5） 其他设置请参考上面内容快速接入。

附录：

WXSample 地址；

https://github.com/xkli/WXSample.git。

8.4.2 集成到 iOS

8.4.2.1 通过 cocoaPods 集成 Weex iOS SDK

首先假设你已经完成了安装 iOS 开发环境和 CocoaPods，具体步骤如下。

1. 添加依赖

导入 Weex iOS SDK 到你已有的项目，可以参考新建项目，在继续下面内容之前，确保你已有的项目目录名称为 Podfile 文件，如果没有，创建一个，用文本编辑器打开。

- 集成 framework。

WeexSDK 在 cocoaPods 上的最新版本可以从这里获取。

在 Podfile 文件中添加如下内容：

```
source 'git@github.com:CocoaPods/Specs.git'
target 'YourTarget' do
   platform :ios, '7.0'
   pod 'WeexSDK', '0.9.5'   ## 建议使用 WeexSDK 新版本
end
```

- 源码集成。

首先拷贝 ios/sdk 目录到你已有项目目录（此处以拷贝到你已有项目的根目录为例子），然后在 Podfile 文件中添加：

```
source 'git@github.com:CocoaPods/Specs.git'
target 'YourTarget' do
   platform :ios, '7.0'
   pod 'WeexSDK', :path=>'./sdk/'
end
```

2. 安装依赖

打开命令行，切换到你已有项目 Podfile，这个文件存在的目录，执行 pod install，没有出现任何错误，表示已经完成环境配置。

3. 初始化 Weex 环境

在 AppDelegate.m 文件中做初始化操作，一般会在 didFinishLaunching WithOptions 方法中做如下添加：

```
//business configuration
[WXAppConfiguration setAppGroup:@"AliApp"];
[WXAppConfiguration setAppName:@"WeexDemo"];
[WXAppConfiguration setAppVersion:@"1.0.0"];

//init sdk enviroment
[WXSDKEngine initSDKEnviroment];

//register custom module and component, optional
[WXSDKEngine registerComponent:@"MyView" withClass:[MyViewComponent
class]];
[WXSDKEngine    registerModule:@"event"    withClass:[WXEventModule
class]];

//register the implementation of protocol, optional
[WXSDKEngine      registerHandler:[WXNavigationDefaultImpl      new]
withProtocol:@protocol(WXNavigationProtocol)];

//set the log level
[WXLog setLogLevel: WXLogLevelAll];
```

4. 渲染 weex Instance

Weex 支持整体页面渲染和部分渲染两种模式，你需要做的事情是用指定的 URL 渲染 Weex 的 view，然后添加到它的父容器上，父容器一般都是 viewController。

```
#import <WeexSDK/WXSDKInstance.h>
- (void)viewDidLoad
{
    [super viewDidLoad];

    _instance = [[WXSDKInstance alloc] init];
    _instance.viewController = self;
    _instance.frame = self.view.frame;

    __weak typeof(self) weakSelf = self;
    _instance.onCreate = ^(UIView *view) {
        [weakSelf.weexView removeFromSuperview];
        [weakSelf.view addSubview:weakSelf.weexView];
    };

    _instance.onFailed = ^(NSError *error) {
        //process failure
    };

    _instance.renderFinish = ^ (UIView *view) {
        //process renderFinish
    };
    [_instance renderWithURL:self.url options:@{@"bundleUrl":[self.url
absoluteString]} data:nil];
}
```

WXSDKInstance 是很重要的一个类，提供了基础的方法和一些回调，如 renderWithURL，onCreate，onFailed 等，可以参见 WXSDKInstance.h 的声明。

5. 销毁 Weex Instance

在 viewController 的 dealloc 阶段，销毁掉 WEEX instance，释放内存，避免造成内存泄露：

```
- (void)dealloc
{
    [_instance destroyInstance];
}
```

8.4.2.2 导入 WEEX SDK framework 到工程

可以通过源码编译出 WEEX SDK，在新的 feature 或者 bugfix 分支，尝试最

新的 feature。

参考此处，直接导入 WEEXSDK。

8.5　使用 WEEXpack 构建移动应用

对于之前使用过 phoneGap、cordova、ionic 等技术的同学来说，需要有一个工具能够帮助开发人员打包出立即能用的 App。这个工具就是 WEEXpack。那 WEEX-tookit 和 WEEXpack 的关系又是什么呢？WEEXpack 是早期的 WEEX 开发辅助工具，是一个项目脚手架、打包工具。随着 WEEX 支持 Vue 之后、WEEX-toolkit 集成了 WEEXpack、WEEX-toolkit，现在也能够帮助我们打包 App，不过是通过调用 WEEXpack 的方式来实现的。另外，WEEX 还有一个插件市场：https://market.dotwe.org，开发人员能够在这个市场里寻找自己需要的插件，同时，也能够上传自己写的插件。本节我们将介绍使用 WEEXpack 来打包生成我们的 App，而不是使用 playground 来预览我们的页面。

使用下面的命令安装 WEEXpack：

```
npm install -g weexpack
```

使用下面命令用 WEEXpack 来创建一个新的工程、之所以不使用之前用 WEEX-toolkit 创建的工程是因为目前，WEEXpack 还不支持打包 WEEX-toolkit 创建的工程。这个能力后续 WEEX 团队会补充上。

```
weexpack create appName
```

通过 create 命令创建的工程默认不包含 ios 和 android 工程模板，创建完成之后就可以切换到 appName 目录下并安装依赖。

```
cd appName && npm install
```

依赖安装好后，输入：

```
weexpack run web
```

我们通过它来起 web 服务，我们知道 WEEX 能够同时构建出支持 web、Ios 和 Android 的代码，run web 意味着我们起的是 web 服务。现在我们可以新增一个 platform，例如 ios platform。

```
weexpack platform add ios
```

安装完后我们就可以开始打包我们的 App 了。

```
weexpack run ios
```

首次打包会安装各种 sdk 依赖，这里需要我们提前把 ios 的开发环境 Xcode 等安装好。除此之外，还需要安装 cocoapods，weexpack 依赖 cocoapods 来安装依赖，如果你之前没有安装过 cocoapods，可能会花费一些时间，因为 cocoapods 首次安装，会做一份巨大的 config 库。由于 Xcode 等 ios 开发环境更新频繁，所以，有的时候，不能一起打包并运行起来，这时候不用慌张，我们可以通过双击 xcworkspace，使用 Xcode 打开该工程。如图 8-6 所示。

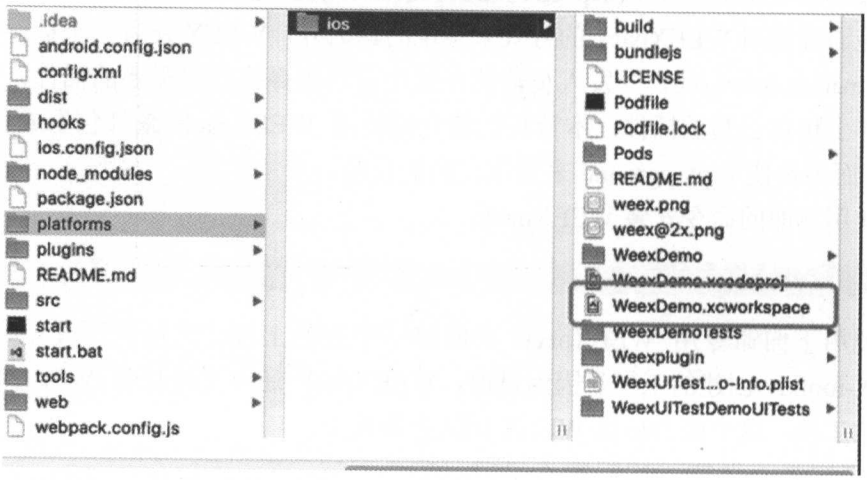


图 8-6

使用 Xcode 进行编译和运行该工程，如图 8-7 所示。

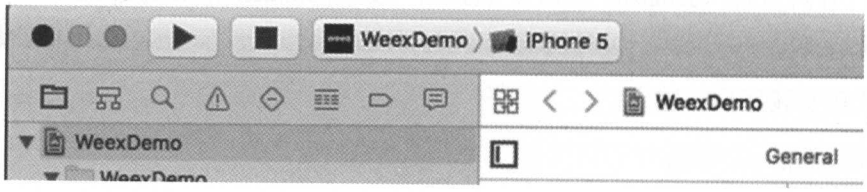


图 8-7

使用 ios 模拟器来运行操作如图 8-8 所示。

图 8-8

这样，我们就成功地将 weex 工程打包成 ios 项目了。

读 者 圈

Vue 前端工程师必备技能
移动开发实战技巧

李利德

百度外卖高级技术经理

历任百度及百度外卖资深工程师

有丰富的互联网项目建设经验和团队管理经验

徐辛承

负责百度外卖用户产品部前端方向

对前端开发架构、技术应用和团队高效协作有自己的见解

擅长用平台化和工程化手段解决问题

策划编辑：张瑞喜
责任编辑：张瑞喜
封面设计：田晨晨

ISBN 978-7-121-33156-5

定价：58.00元